Cache-based Multi-query Optimization for Data-intensive Scalable Computing Frameworks

Pietro Michiardi
EURECOM
michiard@eurecom.fr

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

Sara Migliorini
University of Verona
sara.migliorini@univr.it

ABSTRACT

In modern large-scale distributed systems, analytics jobs submitted by various users often share similar work, for example scanning and processing the same subset of data. Instead of optimizing jobs independently, which may result in redundant and wasteful processing, 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. Extensive experiments on a prototype implementation of our system show significant benefits of worksharing for both TPC-DS workloads and detailed micro-benchmarks.

1. INTRODUCTION

Modern technologies to analyze large amounts of data have flourished in the past decade, starting with general-purpose cluster processing frameworks such as MapReduce [12], Dryad [23] and Spark [42]. 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, in addition to a procedural one. 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.

Currently, such 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 [21] 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 a more efficient resource utilization, and better aggregate performance.

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 [35, 36, 44], materialized views selection [20, 30], and pipelining [11]. Recently, work sharing optimizations operating at query runtime, for staged databases, have also been extensively studied [5, 19, 22, 34]. The idea of reusing intermediate data across queries or jobs running in a distributed environment has also received significant attention: for MapReduce [32, 40], for SCOPE operating on top of Cosmos [38] and for Massive Parallel Processing (MPP) frameworks [15].

In this paper, we study MQO in the context of distributed computing engines such as Apache Spark [42], with analytics jobs written in SparkSQL [4], 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 estima-

.

tion and a knapsack formulation of the problem, that takes into account 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 of the original input set.

We present a prototype of our system built for SparkSQL, and validate it through a series of experiments. First, using the standard TPC-DS benchmark, we provide an overview of query runtime distributions across a variety of different queries which are optimized using our method: overall, our method achieves up to 80% reduction in query runtime, when compared to a setup with no worksharing. Then, we proceed with a synthetic evaluation of individual operators, to clarify which ones benefit most from our technique, including when data is materialized on disk according to different formats. Our main contributions are as follows:

1. We propose a general approach to MQO for distributed computing frameworks that support a relational API. Our approach produces *sharing plans*, that are materialized in RAM, aiming at eliminating redundant work and I/O in a given set of queries.
2. We cast the optimization problem of selecting the best sharing plans as a Multiple-choice Knapsack problem, and solve it efficiently through dynamic programming.
3. Our ideas materialize into a system prototype, which extends the SparkSQL Catalyst optimizer, and that we evaluated extensively, using macro and micro benchmarks. Our results indicate tangible improvements in terms of aggregate query execution times, while fulfilling the memory budget given to the MQO problem.

The rest of the paper is structured as follows. Section 2 covers related work on multi-query optimization. We introduce and formalize our optimization problem in Section 3, and present our methodology in Section 4. In Section 5, we provide the implementation details of our prototype based on SparkSQL. We evaluate the performance of our approach in Section 6. Finally, we conclude the paper in Section 7.

2. RELATED WORK

We now review previous work on MQO, both for traditional RDBMSes and for distributed computing frameworks.

MQO in RDBMSes. Multi-query optimization has been studied extensively [11, 17, 35–37]. More recently, similar subexpressions sharing has been revisited by Zhou et al. in [44], who show that reusable common subexpressions can improve query performance. Their approach avoids some limitations of earlier work [17,35] by (i) considering all generated plans as sharing opportunities to avoid leading to suboptimal plans and (ii) also considering multiple competing covering expressions. More recently, work sharing at the level of the execution engine has been extensively studied [5,19,22,34]. The MQO problem is considered at query runtime, and requires a staged database system. Techniques such as pipelining [22] and multiple query plans [9,10] 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 [20,30,41], whereby the contents of materialized views are precomputed and stored to provide alternative, faster ways of computing a query. Nevertheless, due to the nature of RDBMSes, base relations change frequently, leading to view inconsistency and view maintenance costs. For this reason, a broad range of works addressed the problem of materialized view selection and maintenance, including both deterministic [3,7,35] and randomized [13,25,43] strategies.

In this paper, we focus on analytics queries for systems in which data can be assumed to be static. Such systems are built to support workloads consisting mostly of ad-hoc, long running, scan-heavy queries over data that is periodically loaded in a distributed file system. As such, problems related to view maintenance do not manifest in our setup. Moreover, while materialized views are generally stored permanently (to disk), 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. [38] proposed an extension to the SCOPE query optimizer which optimizes cloud scripts containing common expressions while reconciling physical requirements.

In the context of MPP databases, the work in [15] 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 avoiding redundant processing by batching concurrent MapReduce jobs and make them share some intermediate results was widely studied in [2,8,16,27,28,32,38,40]. The common denominator of such previous work is that they operate at a lower level of abstraction than we currently do in this paper: they analyze low-level programs that use the procedural API to describe an analytical job. For instance, MRShare [32], is a sharing framework for MapReduce that first identifies different jobs sharing portions of identical work. These jobs are then transformed into a compound job such that *scan sharing*, *Map Output sharing* and *Map Functions Sharing* can be achieved.

Caching to recycle work. Finally, we consider previous works [6, 14, 18, 24, 31] that address the problem of reusing intermediate query results, which is cast 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 [14]. Nevertheless, they focus on small problem instances which do not require the general, cost-based approach we present in this work.

3. PROBLEM STATEMENT

This section frames our problem statement and defines a running example that we use in the following sections.

In this paper we focus on the MQO problem only; we gloss over systems aspects that a full-fledged solution should con-

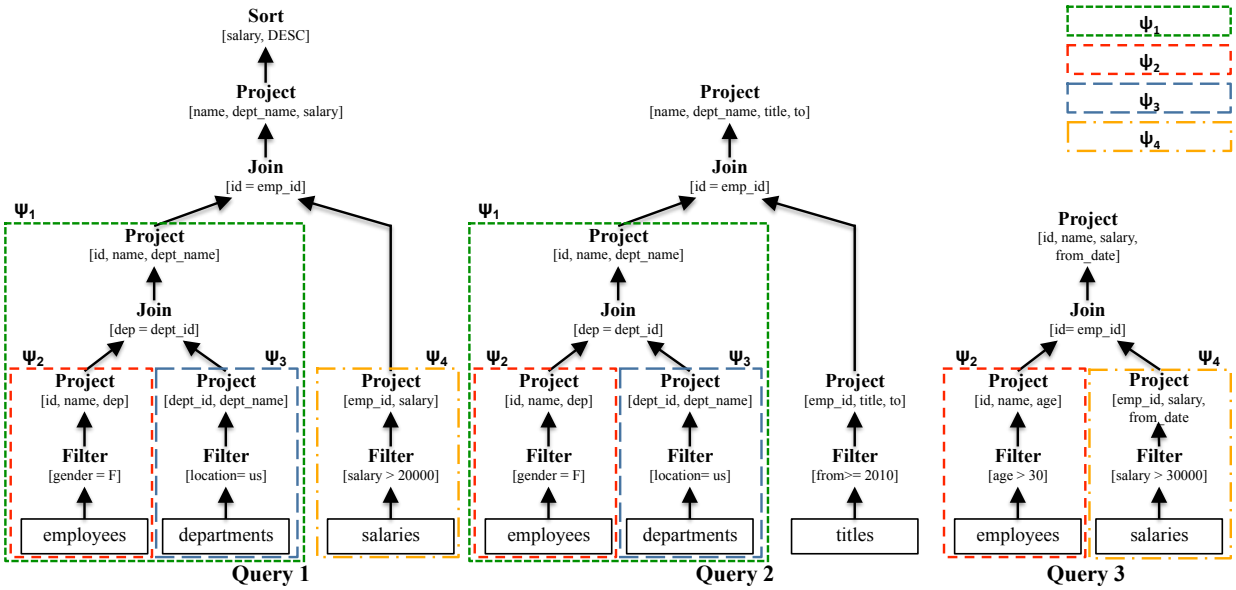

Figure 1: Logical plans for the queries in our running example. Each operator tree has been optimized for query individually. Similar subexpressions (SE) inside logical plans are emphasized by dashed boxes surrounding the corresponding sub-tree of each query logical plan. Boxes with the same border color denotes the same SE.

sider as well. In particular, the assumption that a set of concurrent queries is given as an input to the MQO problem hides some complexity and requires careful engineering. This amounts to the design and management of a queue of pending queries, to determine the queue size, and when (given a sufficient number of queued queries) to trigger the MQO. In addition, given a batch of concurrent queries, in case of low concurrency and sufficient available resources, the system should be able to discern whether to execute such queries in parallel, as in a traditional query-centric model, or to apply MQO, as done for example in [34]. Finally, given an optimized query set, produced by our MQO strategy, the system should determine in which order to schedule the new batch of queries, given a performance objective.

We now introduce a simple running example, that is rich enough to illustrate 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
```

We use Figure 1 to illustrate 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 algebra 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 in RAM distributed relations – which are then 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, joining tables, etc, and materialize intermediate results in RAM, to speed-up query runtime by re-using 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 subtree 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 only once, and the cached intermediate relation could serve three *consumer* queries. To achieve that, we need to build a *covering expression* (CE) that combines the different variants of the predicates appearing in the operators, for instance considering ψ_2 the corresponding CE could be:

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

In a similar vein, the SEs labelled as ψ_3 and ψ_4 share the projection and filtering on **department** and **salaries** relations, respectively.

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. 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. For example, 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. In the following Section, we delve into the detail of our MQO approach.

4. CACHE-BASED WORK SHARING

This section describes our approach to MQO, using a caching operator to materialize in RAM intermediate (distributed) relations that belong to a *sharing plan*. We assume 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:

1. **Similar subexpressions identification.** The goal of this phase is to identify all common and similar subexpressions in the input set, as discussed in Section 4.1. In short, we compute an *operator fingerprint* for each operator in the logical plan of every query and store it in a fingerprint table. Two (or more) operators sharing the same fingerprint constitute a SE. Identified SEs are candidates for building covering expressions (CEs) in the next step.

2. **Building Covering subexpressions, *a.k.a.* sharing plans.** Given all SEs identified in an input set, the goal of this phase is to construct one or more groups of CEs representing candidate sharing plans, as discussed in Section 4.2. Since the search space for building CEs and for their subsequent selection can be very large, in this phase our approach prunes bad SEs, in an attempt to produce few, good CEs candidates.

3. **Sharing plan selection.** The goal of this phase is to select the best combination of CEs, using estimated costs and memory constraints, as shown in Section 4.3. The output of this phase is a series of sharing plans which use *shared operators* (covering those of the underlying SEs) and materialize their output relation in RAM using a cache operator. For this reason, we sometimes refer to such CEs as *caching plans*. We model this step as a Multiple-Choice Knapsack problem, and use dynamic programming to solve it.

4. **Query rewriting.** The last step to achieve MQO is to rewrite the input query set such as to use selected sharing plans, as shown in Section 4.4. Essentially, cached relations pertaining to a CE are *pipelined* to those queries that can be rewritten using that CE. The output of this phase is a new set of rewritten queries, that subsume the input set, although their execution is not guaranteed to be in the original ordering.

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 rule of thumb is also 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 [4]. The natural hierarchy of an optimized logical plan, in general, 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 hierarchy are preferred because they potentially exhibit smaller memory footprints, should their output relation be cached.

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. Thus, 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] In the following, we provide the necessary definitions that are then used to describe the identification of SEs.

Definition 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.*

In the following, 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*) [29], whereby each internal node identifier is the combination of identifiers of its children. More specifically, given an operator u, its identifier, denoted by $\mathrm{ID}(u)$, is given by:

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

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 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 an uncluttered notation, 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* (denoted as ω) is.

Definition 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 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).

Algorithm 1 Similar subexpressions identification

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

```
1: procedure IdentifySEs([τ^Q1, τ^Q2, ...τ^QN])
2:     FT ← ∅
3:     foreach τ ∈ [τ^Q1, τ^Q2, ...τ^QN] do
4:         nodeToVisit ← Add(τ)
5:         while nodeToVisit not empty do
6:             τ^curr ← Pop(nodeToVisit)
7:             ψ ← F(τ^curr)
8:             if CacheFriendly(τ^curr_root) then
9:                 FT.AddValueSet(ψ, τ^curr)
10:            end if
11:            if (! CacheFriendly(τ^curr_root) ∨
12:                ContainsUnfriendly(τ^curr)) then
13:                nodeToVisit ← Add(τ^curr_children)
14:            end if
15:        end while
16:    end for
17:    S ← ∅
18:    foreach ψ ∈ FT.Keys do
19:        if |FT.GetValue(ψ)| ≥ k then
20:            S ← S ∪ FT.GetValue(ψ)
21:        end if
22:    end for
23:    return S
24: end procedure
```

[1] Our method can be easily extended for sharing similar join operators, for example by applying the "equivalence classes" approach used in [44]. 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.

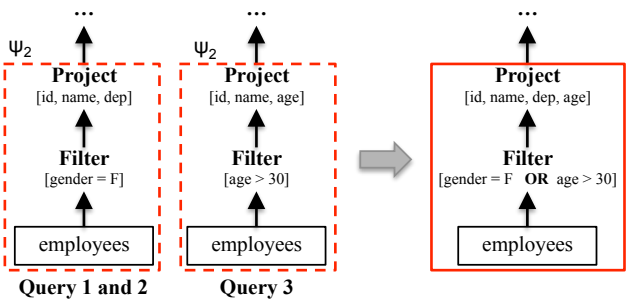

Figure 2: Building covering expression example. The first and second trees are two similar subexpressions. The third tree is the covering subexpression.

At the end, we extract the set of SEs from the HashMap FT: we consider the SEs bigger than a threshold k (*e.g.* with at least two subtrees from two queries) 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 Fig. 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.

4.2 Building Sharing Plans

Given a list of candidate SEs, the goal of this phase is to build 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 his 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 Fig. 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[2]. Fig. 2 illustrates an example of CE for a simple SE of two sub-queries taken from the running example shown in Fig. 1. In particular, we consider the SE labeled as ψ_2.

[2]For the sake of readability, we omit the description of several other optimizations – such as the removal of duplicate predicates – that we have implemented.

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 subtrees in ω_i.

Definition 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 Fig. 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. As in traditional database systems, 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 and their columns. 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 $\mathcal{C}_E(\tau_j)$ 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 $\omega = \{\tau_1, \tau_2, \ldots, \tau_m\}$, the total execution cost $\mathcal{C}(\omega_i)$ related to the execution of all similar sub-trees $\tau_j \in \omega_i$ without the work-sharing optimization is given by

$$\mathcal{C}(\omega_i) = \sum_{j=1}^{m} \mathcal{C}_E(\tau_j). \quad (1)$$

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 ($\mathcal{C}_W$) and retrieving ($\mathcal{C}_R$) costs *associated to the cache operator we use in our approach*, which accounts for write and read operations:

$$\mathcal{C}(\Omega_i) = \mathcal{C}_E(\tau_i^*) + \mathcal{C}_W(|\tau_i^*|) + m \cdot \mathcal{C}_R(|\tau_i^*|), \quad (2)$$

where both $\mathcal{C}_W(|\tau_i^*|)$ and $\mathcal{C}_R(|\tau_i^*|)$ 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 corresponding CE Ω_i.[3]

Given the above costs, 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 corresponding CE Ω_i:

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

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 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.* $w(\Omega_i) = |\tau_i^*| \triangleq |\Omega_i|$.

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 candidate set, but first illustrate the challenges it addresses using the example shown in Figure 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.

[3]In light of the end-to-end MQO process, the last phase amounts to rewrite the queries in the input set to use selected CEs. Such rewrite can introduce additional work, which we currently neglect in our modeling approach: indeed, query rewriting involves highly selective operations, with low cost. This means we assume the dominating cost to be that of reading from RAM, which we found experimentally to be true.

Algorithm 2 Algorithm to generate CE candidates.

Input: Set Ω of CEs
Output: Set of Knapsack items (potential CEs)

```
1: procedure GenerateKPitems(Ω = {Ω_1, Ω_2, ...})
2:     Ω^exp ← ∅
3:     while Ω not empty do
4:         Ω_i ← PopLargest(Ω)
5:         DescSet ← FindDescendant(Ω_i, Ω)
6:         Group_i ← [Ω_i] ∪ Expand(DescSet)
7:         Ω^exp ← Ω^exp ∪ {Group_i}
8:         Remove(DescSet, Ω)
9:     end while
10:    return Ω^exp
11: end procedure
```

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 property is achieved by considering 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 final output of Algorithm 2 then is:

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

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) [39]. 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, $i = 1, 2, \ldots, g$, 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:

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

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 [26]. 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. IMPLEMENTATION DETAILS

In this work, we adopt the Apache Spark [42] processing system and its extension called SparkSQL [4]. As anticipated in Section 3, our prototype implementation glosses over some systems aspects that we discuss next. Queries written in Spark SQL take an abstract form, called a DataFrame. A DataFrame object represents the *logical plan* associated to a query, to produce a given output relation. A logical plan is a tree composed of operators (nodes): thus, each node contains information about the operator type and its attributes (filtering predicates, join columns, etc.).

In Apache Spark, each query issued by a client "lives" within an instance of an individual session. The session implements all the machinery necessary to parse and optimize logical plans, build physical plans and schedule low-level tasks that implement the computation required by a given query. As a consequence, each query runs in "isolation" and sharing work between queries is thus hindered by the very nature of the Apache Spark architecture. To enable worksharing across multiple client queries, it is thus necessary to build a centralized component that can accumulate multiple client queries, optimize them, and schedule their execution. We call this component the *SparkSQL Server*. Additionally, it is necessary to modify the typical Apache Spark workflow to submit an application: individual clients should submit their applications to the SparkSQL Server, by passing the logical plans associated to each query.

In this paper, we focus on the implementation of the algorithmic aspects of worksharing in the SparkSQL Server, because our goal is to validate and assess the benefits of the proposed methodology. Hence, we assume the (locally optimized) logical plans associated to each individual client query to be available in the SparkSQL Server, taking the form of a collection (e.g., a list) of DataFrames. Hence, our prototype implementation materializes as an extension to the existing single query optimizer designed for SparkSQL, namely the Catalyst module [4]. To do so, we follow the optimization process of 4 phases discussed in Sect. 4. Operator fingerprints are computed to identify all similar subexpressions in the first phase. Phase 2 and phase 4 require query transformations, which we achieve using the Scala's pattern

matching and the TreeNode library of Catalyst in SparkSQL. Transforming rules are passed as a function to specify how to transform the logical plan trees. In our prototype, cardinality estimation is achieved by a pre-processing phase, that produces the statistics needed in phase 3.

Finally, once the worksharing optimization process has produced the caching plans according to the solution to the multiple-choice knapsack problem, the SparkSQL Server behaves as a regular Apache Spark client and submits (sequentially) each rewritten client query to the compute cluster.

6. EXPERIMENTAL EVALUATION

We now present experimental results to evaluate the effectiveness of our methodology, which we implement for the Apache Spark and SparkSQL systems.[4] First, we focus on a general overview of the performance gains achieved by our MQO approach, using the standard TPC-DS benchmarking; we then proceed with a detailed analysis of caching efficiency for individual operators and simple queries.

6.1 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.

For the macro-benchmark, 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 [1], and generate a CSV dataset with scaling factor of 50.

For the micro-benchmark, each worker is granted 6 GB of RAM each, of which half is used for caching data. The **synthetic dataset** used for the experiments is stored in HDFS using Parquet and CSV formats. The dataset is a table of 30 columns. The first ten columns $n_i, i = 1, 2, ..10$ are of integer data type, randomly and uniformly generated in the range $[1, 10^{i+2}]$. The next ten columns $d_i, i = 1, 2, ..10$ and the successive ten $s_i, i = 1, 2, ..10$ are of double (in range $[0, 1]$) and string (of length 20) data types, respectively. We also vary the input sizes from 10 millions (10M) records (of size 3GB on disk) to 100M records (of size 30GB on disk). Note that, for clarity of exposition, when we display the query plans and input relations for the micro-benchmark we only show a subset of 5 out of the 30 attributes, and label such attributes with names. Hence, for our examples, we call our input relation "people", where attribute n_1 becomes "age", and attribute s_1 becomes "name", and so forth.

In both benchmarks, we use Apache Spark 2.0. Before running any test, we clear the operating system's buffer cache in all workers and master to obtain more accurate results. We also disable the "compression on caching data" feature of Spark.

6.2 Macro-benchmarks

We begin with a full-fledged performance benchmark, where we use the standard TPC-DS benchmark adapted to the Apache Spark SQL system [1] to evaluate the benefits of our MQO approach. In particular, 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.

[4]Source code of our prototype is available as an open source contribution, available here: `https://github.com/DistributedSystemsGroup/spark-sql-worksharing`

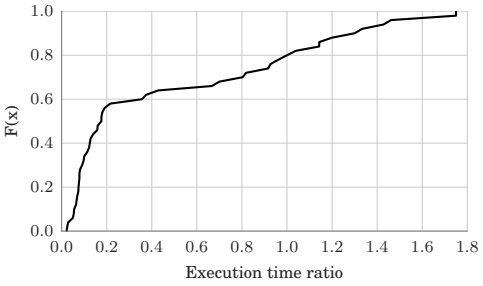

Figure 3: CDF of the performance gains of worksharing for a TCP-DS workload consisting of 50 selected queries.

Next, we present results for a setup in which we consider all the 50 queries and execute them in the order of their identifiers, as established by the TPC-DS benchmark. In other words, this experiment identifies all sharing opportunities in a workload consisting of 50 queries, and applies our worksharing optimization to all of them. 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, as we discuss in Section 6.3. Overall, our optimizer has identified 60 SEs, and it has built 45 CEs. The cache used to store the output of the optimization process 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).

Next, we consider an experimental setup in which we emulate the presence of a queuing component that triggers the execution of our worksharing optimization, as anticipated in Sections 3 and 5. 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 experimental 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 run-

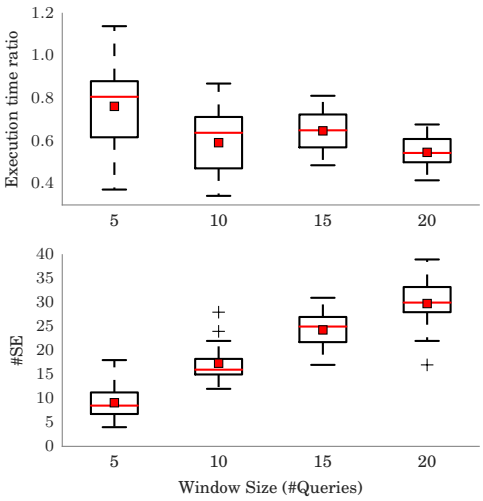

Figure 4: Execution time ratio and number of similar subexpression within a group of queries (given by the size of the window) as the window size increases.

time (median) with a window size of only five queries, which ramps up to 45% when the window size is set to 20 queries.

As anticipated in Section 3, 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.

6.3 Micro-benchmarks

Next, we evaluate our system through a series of experiments based on simple workloads composed by two queries, reading the same synthetic input table, from which we display 5 heterogeneous (numerical and categorical) attributes, for the sake of conciseness. In each experiment, the two queries are run sequentially. We measure and compare the runtime of the Spark jobs associated to each query, according to three strategies: *i)* without worksharing, *ii)* by having the full input relations cached by the system (FC), *iii)* with our worksharing technique (WS). In the latter case, the optimization produces a single CE: hence, the first query triggers the evaluation of the CE, which is cached, whereas the second query benefits the most from worksharing. Each experiment is run three times, and results are averaged.

Overall, our results indicate:

- Roughly 50% and 30% improvement in *aggregate query latencies* for CSV and Parquet files, respectively.
- Our worksharing technique outperforms the naive caching of full input relations.
- Between 25% and 40% less space used by our method with respect to the full cache technique.

In this work we present queries that use *filter* and *project* operators because they appear very frequently in data analysis, where such operators are usually pushed as close as possible to the input tables by traditional query optimization techniques. We also consider the *join* operator that

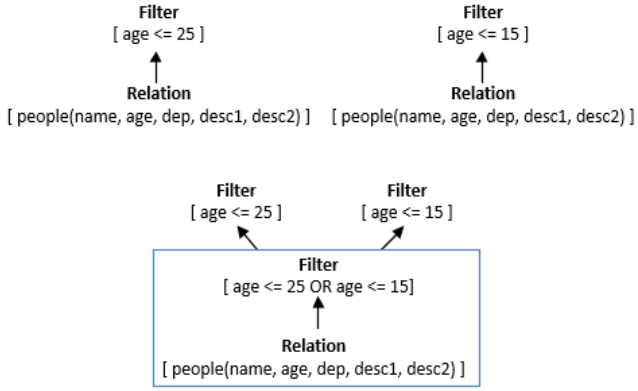

Figure 5: Query (top) and Cache (bottom) plans for Filter-based queries. The input relation is derived from the synthetic dataset, with a labelled schema used for illustration.

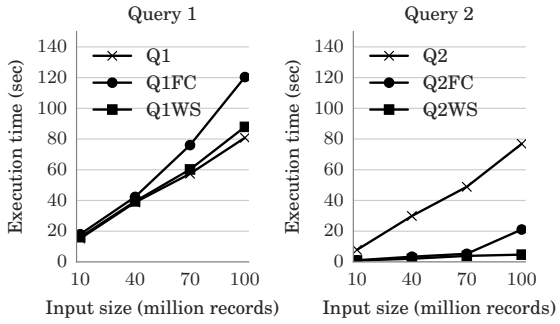

Figure 6: Micro-benchmark for filter-based queries: individual query latencies, comparing baseline, naive caching and our worksharing as a function of input size.

involves shuffling data across the network, but defer discussion to the end of the Section.

Filter-based queries

We consider two simple queries whose logical plan is depicted in Figure 5 (top). Both queries read data from the same input relation, and apply a filter operator with two different predicates on the same attribute. The logical plan displayed at the bottom of the Figure is the output of our multi-query optimizer, that is, a single CE covering both input queries. In this case, the optimized CE can be manually verified: it reads data from the input relation, applies a filter operator with a combined predicate (using the `OR` logical operator), and caches the intermediate result.[5] The output of each query can be obtained by applying the filter specific predicate on the cached result.

Next, in Figure 6, we show the individual query runtime, without any optimization (Q_i), using the caching option (Q_iFC) and with our optimization (Q_iWS) for CSV input files. In the baseline case, without caching nor worksharing, both queries have similar execution times, that grows linearly with the input size. Instead, when naive caching is used (Q_iFC), individual query runtime change: it increases

[5]Note that the `cache()` operator in Apache Spark is a transformation. As a consequence, it takes effect only upon the first call to an action, with the first (rewritten) query. Thus, the first query effectively "pays the price" for caching.

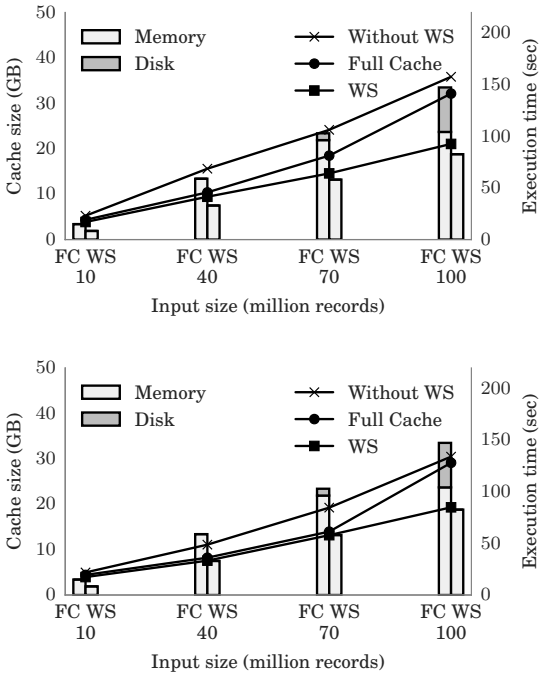

Figure 7: Micro-benchmark for filter-based queries: aggregate query latencies and memory utilization with CSV input format (top) and Parquet input format (bottom).

quite dramatically for Q_1FC, because this query incurs the overheads associated with a cache operation on the entire input relation, and it decreases for Q_2FC, because this query benefits from a cached input. For large input sizes, however, the system spills cached data on disk, with reduced benefits on execution times. Our MQO approach, instead, achieves superior performance: Q_1WS pays a small price for caching just what is needed to produce the output relations, whereas Q_2WS runtime becomes negligible and is marginally affected by the input relation size.

The aggregate latency of the workload is shown in Figure 7 (for both CSV and Parquet input format). Our worksharing strategy consistently obtain 40%-50% aggregate latency improvement with respect to the baseline. Figure 7 shows also the system memory utilization dedicated to caching. A naive caching strategy that stores the entire input relation suffers from capacity constraints: once the available RAM is depleted, data is spilled to disk, as visible when the input size increases. Instead, with our worksharing approach, we only cache the output of the combined filtering operation defined by the CE, while satisfying capacity constraints: the RAM is wisely used and spilling to disk is not required. Overall, the cache size is roughly 25% of the size of input data size, for both the Parquet and CSV case. Our multi-query optimizer uses only 2/3 of the caching capacity (25% of 200M records costs 16GB of cache size, which totals 24GB in our system, that is half of 6GB times 8 workers).

Projection-based queries

Next, we consider simple queries that only perform projection operations, as shown in Figure 8. Both queries read data from the same input relation, and apply a project operator on a set of different attributes (top of the Figure).

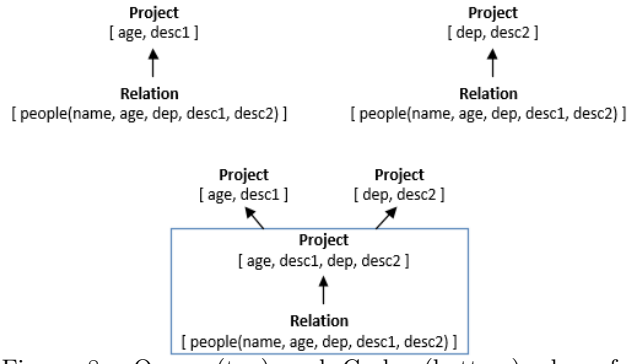

Figure 8: Query (top) and Cache (bottom) plans for Projection-based queries, using synthetic data.

The logical plan displayed at the bottom of the Figure is the optimized CE covering both input queries. The CE can be manually verified: it reads data from the input relation, applies a project operator with the union of each individual query attributes, and caches the intermediate result.

Note that projection-based queries can benefit from advanced data representations such as Parquet. In particular, Parquet is geared toward columnar storage: thus, we expect projection queries to execute efficiently, because they only read from disk the data pertaining to selected attributes. It is thus interesting to verify the benefits of caching, if any, for such workloads. In the next series of results, we focus on the Parquet input data types, as shown in Figure 9.

Overall, our results share similar qualitative considerations as for filter-based queries: the price to pay to cache data affects the latency of Q_1FC and, to a lesser extent, of Q_1WS. However, the benefits from caching in Q_2FC and Q_2WS are less pronounced than for filter-based queries. This is due to the efficiency (for projections) of the Parquet data format. Also, our MQO method is superior in its use of the RAM, when compared to a naive caching strategy.[6]

As shown in Figure 9 (bottom), the aggregate performance for a simple projection-based workload is in favor of our approach, when compared to naive caching. Nevertheless, we measure no tangible benefit when compared to the baseline: the execution time "lost" by query 1 to invest in caching does not pay enough benefits for query 2. Obviously, results obtained with CSV data files are largely in favor of our approach, which outperforms the others in all metrics. Indeed, the parsing costs required for projections translates in non-negligible CPU costs, in addition to increased disk I/O costs due to the requirement to read all data in the input relation.

Other types of queries: discussion and summary

We have conducted an extensive experimental campaign considering queries that combine projection, filtering, as well as join operators. In fact, the generation of synthetic queries to emphasize different aspects related to system performance

[6]The attentive reader might have noticed that also our method eventually spills some contents of the cached data to disk. This is explained by two effects: *i)* Apache Spark dynamically adjusts at runtime the amount of memory dedicated to store cached data, and thus overrides the 50% setting we use in our experiments; *ii)* our methodology is based on cardinality estimation to compute the weight of a CE: as a consequence, estimation errors might induce the system to spill some records on disk.

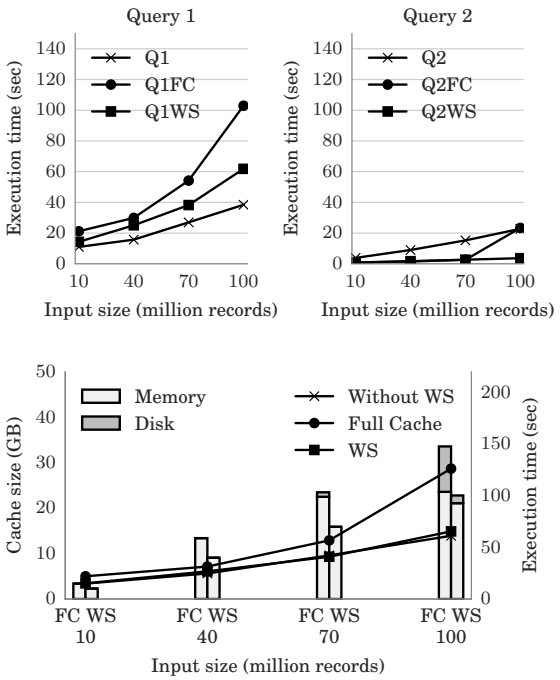

Figure 9: Micro-benchmark for Projection-based queries using Parquet input format: individual query latencies (top), aggregate query latencies and memory utilization (bottom).

and the impact of our optimization mechanism, can yield a combinatorial amount of results that we summarize below.

In general, when using the CSV input format, the benefit of worksharing is remarkable: indeed, this wide-spread input data format stresses underlying parallel processing frameworks quite heavily, especially when it comes to parsing, tokenization and type casting costs, as for example discussed in [33]. Thus, caching achieves consistent gains in query execution time, because it can also save over the "hidden" costs related to a specific data format. Conversely, using the Parquet data format can attenuate the benefits of caching, and this is true mainly for simple queries with projections, as discussed above. Instead, when queries involve operators such as joins, for example, other quantities (such as network I/O) are predominant over the savings in data access times, and the effects of caching become tangible again.[7]

Another way to judge our micro-benchmark results, is to view the query variants we propose as ways to assess different aspects of the cost model defined in this work, and consequently the suitability of the knapsack formulation of the optimization problem. We remark that our results are very robust with respect to the constants we use in our cost model, which is truly desirable as it implies that little to no tuning is required to use our method.

7. CONCLUSION

Complex queries for analyzing massive amounts of data have become commonplace today: such trend has been fueled by several efforts to support SQL capabilities on top of large-scale distributed processing frameworks. Similarly to what happens in traditional relational database management systems, users share access to data-intensive processing frameworks and induce workloads with a high degree of redundancy in terms of queries containing similar (sub)expressions. As a consequence, the traditional problem of *multi-query optimization* that has been largely studied for RDBMSes, also apply to recent data processing frameworks.

In this paper, we presented a new approach to multi-query optimization that uses in-memory caching primitives to improve the efficiency of data-intensive, scalable computing frameworks, such as Apache Spark. Our methodology takes as an input a batch of queries written with the SparkSQL API, and analyzes them to find common (sub)expressions, leading to the construction of an alternative execution plan based on covering expressions, that subsume the individual work required by each query. To make the search problem tractable and efficient, we have used several techniques including: modified hash trees to quickly identify common sub-graphs, and an algorithm to enumerate (and prune) feasible common expressions. We then cast the multi-query optimization problem as a multiple-choice knapsack problem: each feasible common expression is 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 is to fill a knapsack of a given capacity (representative of memory constraints) optimally.

To quantify the benefit of the proposed methodologies, we implemented a prototype of our method for Apache Spark SQL, and we designed two families of experiments. First, we used the well-known TPC-DS workload to design a macro-benchmark on realistic queries and data. Our results indicated that worksharing opportunities are frequent, and that our proposed methodology brings substantial benefits in terms of reduced query runtime, with up to an 80% reduction for a large fraction of the submitted queries. Then, using micro-benchmarks, we studied the benefits (and costs) of worksharing for a workload of simple queries, focusing on individual relational operators, including filter-based and projection-based queries.

In our research agenda, we will consider systems aspects related to the management of a queuing mechanism to accumulate submitted queries in a batch window, and analyze the trade-off that exist between query runtime and execution delay. In this paper, we have obtained promising preliminary results, showing that even small window sizes (that would mitigate execution delays) are sufficient to reap the benefits from worksharing.

8. ACKNOWLEDGMENTS

The authors would like to thank Khoa Nguyen Trong, Duy-Hung Phan and Quang-Nhat Hoang-Xuan for the valuable discussions and contribution in the early stage of this work.

[7]Data compression techniques can be helpful in this case, but we defer their analysis to future work.

9. REFERENCES

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

[2] P. Agrawal, D. Kifer, and C. Olston. Scheduling shared scans of large data files. *VLDB Endowment*, 1(1):958–969, 2008.

[3] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in sql databases. In *VLDB*, volume 2000, pages 496–505, 2000.

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

[5] 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*, pages 519–530, 2010.

[6] T. Azim, M. Karpathiotakis, and A. Ailamaki. Recache: Reactive caching for fast analytics over heterogeneous data. *VLDB Endowment*, 11(3), 2017.

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

[8] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin. Incoop: Mapreduce for incremental computations. In *Proc. of ACM SoCC*, page 7, 2011.

[9] G. Candea, N. Polyzotis, and R. Vingralek. A scalable, predictable join operator for highly concurrent data warehouses. *VLDB Endowment*, 2(1):277–288, Aug. 2009.

[10] G. Candea, N. Polyzotis, and R. Vingralek. Predictable performance and high query concurrency for data analytics. *The VLDB Journal*, 20(2):227–248, Apr. 2011.

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

[12] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Comm. of the ACM*, 51(1):107–113, 2008.

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

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

[15] 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.*, 8(12):1704–1715, 2015.

[16] I. Elghandour and A. Aboulnaga. Restore: reusing results of mapreduce jobs. *VLDB Endowment*, 5(6):586–597, 2012.

[17] S. Finkelstein. Common expression analysis in database applications. In *Proc. of ACM SIGMOD*, pages 235–245, 1982.

[18] 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*, pages 321–333, 2016.

[19] G. Giannikis, G. Alonso, and D. Kossmann. Shareddb: Killing one thousand queries with one stone. *VLDB Endowment*, 5(6):526–537, Feb. 2012.

[20] J. Goldstein and P.-Å. Larson. Optimizing queries using materialized views: a practical, scalable solution. In *ACM SIGMOD Record*, volume 30, pages 331–342, 2001.

[21] 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*, volume 10, pages 1–8, 2010.

[22] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. Qpipe: A simultaneously pipelined relational query engine. In *Proc. of ACM SIGMOD*, pages 383–394, 2005.

[23] 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*, volume 41, pages 59–72, 2007.

[24] 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*, 35(4):24, 2010.

[25] P. Kalnis, N. Mamoulis, and D. Papadias. View selection using randomized search. *Data & Knowledge Engineering*, 42(1):89–111, 2002.

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

[27] 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*, pages 985–996, 2011.

[28] B. Li, E. Mazur, Y. Diao, A. McGregor, and P. Shenoy. Scalla: a platform for scalable one-pass analytics using mapreduce. *ACM Trans. on Database Systems*, 37(4):27, 2012.

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

[30] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham. Materialized view selection and maintenance using multi-query optimization. In *ACM SIGMOD Record*, volume 30, pages 307–318, 2001.

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

[32] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. Mrshare2: Sharing across multiple queries in mapreduce. *VLDB Endowment*, 3(1-2):494–505, Sept. 2010.

[33] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun. Making sense of performance in data analytics frameworks. In *Proc. of USENIX NSDI*, pages 293–307, 2015.

[34] I. Psaroudakis, M. Athanassoulis, and A. Ailamaki. Sharing data and work across concurrent analytical queries. *VLDB Endowment*, 6(9):637–648, July 2013.

[35] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *ACM SIGMOD Record*, volume 29,

pages 249–260, 2000.

[36] T. K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, Mar. 1988.

[37] J. Shim, P. Scheuermann, and R. Vingralek. Dynamic caching of query results for decision support systems. In *Proc. of IEEE SSDBM*, pages 254–, 1999.

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

[39] P. Sinha and A. A. Zoltners. The multiple-choice knapsack problem. *Operations Research*, 27(3):503–515, 1979.

[40] G. Wang and C.-Y. Chan. Multi-query optimization in mapreduce framework. *VLDB Endowment*, 7(3):145–156, Nov. 2013.

[41] J. Yang, K. Karlapalem, and Q. Li. Algorithms for materialized view design in data warehousing environment. In *VLDB*, volume 97, pages 25–29, 1997.

[42] 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*, pages 2–2, 2012.

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

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