m2r2: A Framework for Results Materialization and Reuse in High-Level Dataflow Systems for Big Data

Vasiliki Kalavri
School of Information and Communication Technology
KTH Royal Institute of Technology
Stockholm, Sweden
kalavri@kth.se

Hui Shang
School of Information and Communication Technology
KTH Royal Institute of Technology
Stockholm, Sweden
hshang@kth.se

Vladimir Vlassov
School of Information and Communication Technology
KTH Royal Institute of Technology
Stockholm, Sweden
vladv@kth.se

***Abstract*—High-level parallel dataflow systems, such as Pig and Hive, have lately gained great popularity in the area of big data processing. These systems often consist of a declarative query language and a set of compilers, which transform queries into execution plans and submit them to a distributed engine for execution. Apart from the useful abstraction and support for common analysis operations, high-level processing systems also offer great opportunities for automatic optimizations. Existing studies on execution traces from big datacenters and industrial clusters show that there is significant computation redundancy in analysis programs, i.e., there exist similar or even identical queries on the same datasets in different jobs. Furthermore, workload characterization of MapReduce traces from large organizations suggest that there is a big need for caching job results, that will enable their reuse and improve execution time.**

In this paper, we propose m2r2, an extensible and language-independent framework for results materialization and reuse in high-level dataflow systems for big data analytics. Our prototype implementation is built on top of the Pig dataflow system and handles automatic results caching, common sub-query matching and rewriting, as well as garbage collection. We have evaluated m2r2 using the TPC-H benchmark for Pig and report reduced query execution time by 65% on average.

***Keywords*—results reuse; materialization; computation redundancies;**

I. Introduction

Big data collection, processing and analysis is becoming one of the major concerns for large and small organizations, companies and academic institutions. Operations, business decisions, product recommendations and numerous other everyday tasks are increasingly relying on processing and analyzing large datasets of diverse formats and heterogeneous sources. The need for making the power of big data analysis available to non-experts and analysts with no programming experience, quickly led to the development and adaptation of high-level, dataflow systems for data analysis.

Pig [1], Hive [2] and Jaql [3] are among the most widely used high-level dataflow frameworks for big data analytics. They offer an easy programming experience, using declarative languages and support for common data analysis operations, such as filtering, projection, join, grouping, etc. Their advantages include ease of use, fast prototyping, readable and easily-maintainable programs. Studies show that a big percentage of analytics is performed using such high-level layers [4]. Apart from their obvious benefits for users, high-level dataflow systems also offer great opportunities for automatic optimizations.

In this paper, we propose an optimization regarding identifying and avoiding computational redundancies, i.e., similar or identical queries in different jobs. It is based on recent research studies revealing that redundancies exist in a big extend among typical analysis workloads [5], [6], [7], showing a 30-60% of similarity in queries submitted for execution. In other words, parts of queries or even whole jobs submitted for execution re-appear unchanged in future job submissions. A common case is some kind of initial filtering or transformation on a dataset before the main analysis task. For example, filtering out badly formatted records, spam e-mails or transforming a data representation into another.

The common way to implement this optimization is by caching query and sub-query results, to avoid re-computing identical tasks in future jobs. The idea is based on the materialized view technique, popular in relational databases [8], [9], [10]. In order to exploit materialized results, the system needs to identify the redundancy in future job submissions and provide a mechanism to rewrite queries, so that stored results can be reused. In the context of high-level dataflow systems, there has been some remarkable previous work, mainly for DryadLINQ and Pig [7], [5], [6], [11]. The proposed materialization and reuse frameworks handle the problem by providing an execution plan matcher and query rewriter. These frameworks are naturally highly coupled to the underlying processing systems and execution engines. ReStore [11], for example, matches plans at the physical level, assuming that Pig scripts are translated into MapReduce jobs, while DryadInc [6] assumes a Dryad DAG as the execution model.

However, there has been recent interest in integrating popular high-level dataflow languages with alternative execution engines or developing new ones. In particular, Shark [12] offers an implementation of Hive on top of Spark [13] and PonIC [14] translates Pig scripts into Stratosphere [15] jobs. Furthermore, several different parallel-dataflow systems can be used by the same organization, inside the same datacenter or cluster to process common datasets. Since all these technologies are quite recent and are still evolving, it might be the case that different development teams, inside the same company, use different frameworks and languages to implement similar analysis tasks and process common datasets.

We are, therefore, in need of a language and execution engine-independent and extensible framework for storing, managing and re-using query results. We observe that, despite the differences in the backends of existing high-level dataflow processing systems, they do share common design characteristics on higher layers. These systems often offer a high-level declarative-type language, whose statements correspond to data operators, defining how data will be organized and processed. Thus, a query is normally translated into a DAG of operators, called the *logical plan*. This plan is then optimized and translated into a *physical execution plan*, which defines how the operators will be implemented and executed on the underlying parallel execution engine. For example, in the case of Pig and Hive, the final plan is a DAG of MapReduce jobs, while in the case of Shark, it is a Spark dataflow and in the case of PonIC it is a Stratosphere job.

In this paper, we present m2r2 (materialize-match-rewrite-reuse), a language-independent and extensible framework for storing, managing and using previous job and sub-job results. In order to achieve generality and support different languages and backend execution engines, we have chosen to base our design at the logical plan level. We provide a detailed description of general techniques for identifying candidate sub-plans for materialization. We propose mechanisms for efficient storage and retrieval of plans and sub-plans, in order to exploit reuse opportunities and we also discuss garbage collection and management of the results repository. We report our on-going work on a prototype implementation using the Pig framework and MySQL Cluster as the repository for storing and managing plans and sub-plans. We present very promising preliminary results, using the TPC-H benchmark for evaluation. The main contributions of this paper are as follows.

- We discuss and identify architecture and design similarities in high-level dataflow processing systems for big data, in order to find the proper level for providing a reuse mechanism based on materialization an caching of previous query results.
- We propose a language and execution engine-independent framework, m2r2, for results materialization and reuse.
- We describe a prototype implementation of our materialization and reuse framework for the Pig system.
- We provide an evaluation of our prototype implementation of m2r2, using the TPC-H benchmark for Pig.

The rest of this paper is organized as follows. Section II provides a brief overview of the materialized view techniques in relational databases. Section III discusses the similarities in the design and implementation of popular high-level dataflow systems for big data. Section IV presents the design of m2r2 and Section V discusses implementation details regarding our prototype on Pig. In Section VI, we provide evaluation results. Section VII discusses related work, while we conclude and share our future research directions in Section VIII.

II. Background

A. Materialized Views in Relational Databases

A materialized view in the context of relational databases is a derived relation, stored in the database. Creating and managing materialized views is driven by several applications, such as query optimization, maintaining physical data independence, data integration and others [16]. In this work, we are only interested in materialized views techniques used for query optimization. The idea is based on the fact that queries can be computed from materialized views, instead of base relations, by reusing results of common sub-queries. We briefly discuss the three main problems related to this technique: view design, view maintenance and view exploitation.

1) View Design: View design determines which views will be materialized, considering the trade-off between the limited space for storing the views and the search cost for finding a related view. Thus, it is not practical to create a view for every sub-query, but instead, use a technique to select which sub-queries to materialize. View design is carried out in two steps, *view enumeration* and *view selection*. View enumeration aims to reduce the number of candidate views to be considered by the selection phase, by filtering out the non-related views. View selection is based on a cost-benefit model. A view is considered beneficial if it is expensive to compute and if it can be reused by other queries. The cost is computed based on the overhead to select, create and store the views and the overhead to keep the views updated. Since relational databases use cost-based query optimizers, View selection can be easily integrated with the query optimizer. Therefore, views are selected by the query optimizer based on their benefit and cost.

2) View Maintenance: View maintenance refers to the problem of updating the materialized views when the base relations change. Specifically, when operations such as Insert, Update or Delete are performed on the base relations, the materialized views get "dirty" and they should either be updated or garbage collected. [17] discusses the view maintenance problems and techniques in detail. We are not interested in view maintenance, since most parallel processing platforms assume append-only input data. However, we discuss garbage collection in Section IV-E.

3) View Exploitation: View exploitation describes how to efficiently use materialized views for query optimization. It includes two phases, *view matching* and *query rewriting*. View matching defines how to find the related views that can be used for answering queries. Query rewriting generates a new query, using the selected views. The rewritten query can either be an equivalent expression to the original and provide an accurate answer or, only provide a maximal answer. In this paper, when referring to query rewriting, we always mean equivalent rewriting.

III. Design and Similarities of High-Level Dataflow Systems for Big Data

In this section, we summarize the system design and common characteristics of popular high-level dataflow systems for big data analytics. Our study is mainly based on Pig [1], Hive [2], Jaql [3] and DryadLINQ [18]. We discuss the main system components and focus on the architecture and compilation similarities, which motivate our design decisions.

A. Language Layer

The majority of high-level dataflow processing systems offer a declarative, SQL-like scripting language for writing applications. Programs consist of series of statements, each of which, defines a transformation on one or more collections of datasets and produces new collections. Data is usually read from a distributed file system and final results are also stored there. All systems allow user-defined functions, which can be used in conjunction with the language-provided statements. A wide variety of datatypes is also supported, allowing schema specification and nested data structures.

B. Data Operators

The language statements correspond to *data operators*, objects that encapsulate the logic of the transformations to be performed on datasets. Data operators have one or multiple inputs and usually one output. We refer to the special operator which accepts input from persistent storage, such as a file system, as the *Load* operator and to the special operator that writes its output to persistent storage as the *Store* operator. We refer to common analysis operators for manipulating collections of data such as Filter, Group By, Join, Order, etc as *Relational* operators. Relational operators are distinguished from *Expression* operators, such as Sum, Count, Avg, (elsewhere also seen as functions), which can be composed to form expressions and are usually nested inside relational operators. Some high-level languages also support *control-flow* operators, which we do not consider at present.

C. The Logical Plan

After a script is submitted for execution, it is sent to the parser, which is responsible for creating the Abstract Syntactic Tree (AST). If there are no syntax violations, the AST is transformed into a Directed Acyclic Graph (DAG), called the *logical plan*. The nodes of this plan correspond to data operators, which are connected by directed edges, denoting data flow. The logical plan usually has one or more sources, corresponding to Load operators and at least one sink, which corresponds to the Store operator.

D. Compilation to an Execution Graph

In order to produce the final execution plan, the logical plan goes through two main phases, optimization and translation. During the optimization phase, the plan is simplified and data-flow optimizations are applied, such as filter pushdown, column pruning, etc. The goal is mainly to reduce the amount of data that will be transfered from one operator to the other or create parallelization opportunities. The output of the optimization phase is a rewritten, optimized logical plan. The optimized logical plan is next transformed into a lower-level representation, usually referred to as the *physical plan*. The physical plan is a more detailed and usually larger DAG of fine-grained operators, the *physical* operators. While the logical operators only contain information about the semantics of an operation, the physical operators encapsulate information regarding its actual physical execution. For example, a logical operator Join represents the result of matching two or more datasets based on a key and collecting the set of records that are matched. On the other hand, a physical operator Join contains information about the specific execution strategy that will be used for executing the join (hash-based approach, a merge-sort, the replicated strategy, etc.). Finally, the physical plan is translated into an engine-specific dataflow DAG of tasks or jobs, such as Map-Reduce jobs or Dryad jobs, each encapsulating a part of the initial logical plan.

E. Discussion

We observe that all of the systems share very similar designs on the upper layers, from the language layer until the optimization of the logical plan. The step of translating the logical plan into a physical plan is where the system logics start to divert. This is mainly due to the differences that exist in the backend frameworks, since physical operator implementations essentially depend on the capabilities of the underlying execution engine. We, therefore, believe that it is not a wise design choice to make any assumptions about the backend execution engine. Instead, since our main goal is to make a general framework, we build the materialization and reuse framework at the logical level, which is more stable and less likely to change. The logical plan level is abstract enough to provide us with some information about operator costs, even if we might lose some reuse opportunities, due to reduced granularity. It is true that the more we move down in the compilation process layers, the more fine-grained operators we can exploit, with higher chance for reuse and more information regarding operator costs. However, in that case, we would have to build a specialized system for each language and execution engine. It is our goal to be able to exploit reuse and sharing opportunities among different frameworks, as well as provide support for adding new operators and new languages in the future. We are certain that the logical plan layer is the appropriate point for integration, in order to build a language-independent and execution-engine independent, extensible and configurable framework for results materialization and reuse.

IV. m2r2 Design

We summarize here our main design goals.

- *Independence of the high-level language.* Our design is based on the assumption that there exists an abstract logical operator layer, similar to the one described in Section III-B. If necessary, this layer can be customized based on the different characteristics and requirements of each language.
- *Independence of the execution engine.* This design goal is fulfilled by choosing the logical layer for plan matching. Query rewriting happens before the compilation into physical operators and is, therefore, independent of their implementation.
- *Extensibility.* It is our intention to create a fully extensible framework, so that new operators, languages and execution engines can be easily supported.
- *Configurability*. Our preliminary experiments show that system parameters are very sensitive to workload characteristics. Therefore, we have decided to allow the users to tune important system parameters, such as the set of operators after which sub-plans are materialized, the degree of replication of stored results and the garbage collection policy.
- *Effectiveness and Efficiency.* The reuse mechanism should provide some gain in query execution and this

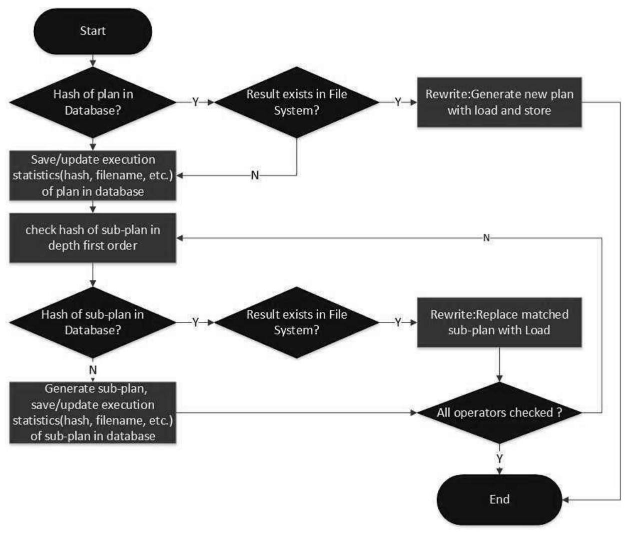
Start
Hash of plan in Database?
Y
Result exists in File System?
Y
Rewrite:Generate new plan with load and store
Save/update execution statistics(hash, filename, etc.) of plan in database
N
check hash of sub-plan in depth first order
N
Hash of sub-plan in Database?
Y
Result exists in File System?
Y
Rewrite:Replace matched sub-plan with Load
N
Generate sub-plan, save/update execution statistics(hash, filename, etc.) of sub-plan in database
All operators checked ?
Y
End

Fig. 1: Flow Diagram of the Match and ReWrite Algorithm

gain should exceed the overhead associated with the reuse mechanism.

The five components of our framework, Plan Matcher and Rewriter, Logical Optimizer, Results Cache, Plan Repository and Garbage Collector, are discussed next.

A. *Plan Matcher and Rewriter*

The Plan Matcher and Rewriter takes as input an optimized logical plan and produces a rewritten logical plan, as output, compatible with the logical plan that the compiler of the underlying framework is expecting. It stores a representation of the (sub) plan and execution statistics about the job in the repository. The Match phase performs lookups in the repository for finding potential matches. If the fingerprint of the input plan or its sub-plans exists in the repository and the output exists in the results cache, then we have a match. In this case, the Rewrite phase is activated and generates new logical plan by adding or removing operators from/to the original logical plan. The tasks performed by this component can be summarized in the four following steps: (1) Choose a set of sub-plans from the input plan to materialize; (2) Calculate the fingerprints of the plans and its sub-plans; (3) Store the fingerprints and execution statistics of the selected plan in the repository, if no match is found or rewrite the query if there is a match. A flow diagram of the Plan and Rewrite Algorithm is shown in Figure 1.

1) Sub-Plan Selection Criteria: One of the main challenges we need to solve is selecting which sub-plans to materialize, since each sub-plan has a different cost and possibilities to reoccur in a future job submission. In order to select the most beneficial sub-plans, we examine the following three criteria: (1) Whether the data size of the output is reduced; (2) whether the computation is expensive; (3) reuse possibilities from future queries. Commonly used operators, such as Filter, often used in the very beginning of the logical plan are good candidates for reducing the data size. Projection is another good candidate, as it often performs column pruning. On the other hand, Group and Join are two commonly used expensive operators, which, for some execution engines, such as MapReduce, incur significant I/O overhead. Thus, materializing these operators is also a good idea. In general, users should better combine the selection strategy with a specific workload.

B. *Plan Optimizer*

The logical optimizer optimizes the rewritten logical plan. First, Store operators are inserted in order to generate the sub-plans. Second, the matched sub-plan is replaced by a Load operator, which might not be able to recognize the type of the input data. We must therefore translate the new data scheme. Most of the high-level dataflow frameworks already provide a logical optimizer layer. We highly encourage users to use the provided optimizer or extend it, in case some additional functionality is necessary.

C. *Results Cache*

The Results Cache stores intermediate results and results of whole jobs. In order to simplify implementation, we suggest using the already provided data storage by each framework. e.g. a distributed file system. This way, rewriting plans is greatly simplified and reuse becomes transparent to the underlying system. However, since most systems use replication for fault-tolerance, using the same configuration for storing intermediate results might increase the overhead. We, therefore, encourage users to disable replication for the results cache.

D. *Plan Repository*

The Plan Repository stores the hash of the selected plan, the output path of the plan and statistics about the stored results, such as the reuse frequency and the last access timestamp. It can be implemented as a key-value store or an in-memory relational database.

E. *Garbage Collector*

The Garbage Collector operates based on information collected by the Repository manager and the Results Cache. It locates obsolete records in the Repository and deletes the related data in both the Repository and the Cache. The garbage collection policy is configurable and can be implemented as least-recently-used, based on reuse frequency or a combination of statistics. Garbage collection can be implemented as a periodical background process or can be explicitly invoked by the user. Alternatively, several threshold values can be set, so that garbage collection is triggered when one of them is violated, for example if the available disk space is too low.

V. IMPLEMENTATION

We implemented a prototype reuse framework on top of Pig/Hadoop. The system architecture is shown in Figure 2.

A. *Match and Rewrite Phase*

We have chosen to represent logical plans simply as Strings. The choice is driven by two facts. First, Pig already provides a method for transforming an operator into a String, containing all necessary information for identifying the operator. Therefore, composing operators becomes as simple as concatenating Strings. Second, having the plans in a String representation greatly simplifies the computation of their fingerprints, which we need to store in the repository.

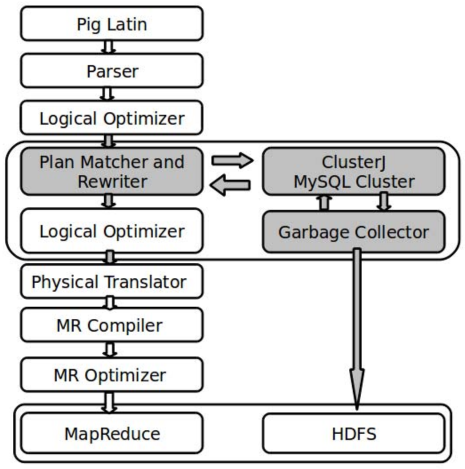

Fig. 2: Implementation of the Reuse Framework on Pig

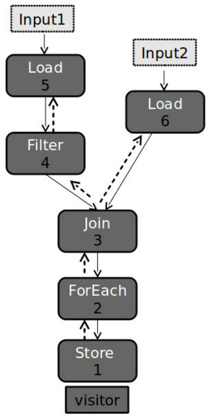

Fig. 3: Bottom-Up Logical Plan Traversal

In Pig, a query can be uniquely identified by its input and its logical plan. For simplicity, we assume that input files are identified by file names and file paths and that these do not change. Therefore, two identical files with different names are treated as different files by our implementation. To calculate the hash of a sub-query, we first calculate the fingerprint of its logical plan, then we calculate the fingerprint of its input and then use their concatenation to uniquely identify the query. We compute the fingerprint of a logical plan with the following steps: (1) A depth-first transversal to retrieve a list of ordered operators in the plan, as demonstrated in Figure 3. (2) Concatenation of the string representations of the operators; (3) Calculation of the hash of the final string. We extend Pig's *getSignature()* method in the *LogicalPlan* class to also acquire the fingerprints of the plan's inner plans during the traversal.

B. *Results Cache and Repository*

We have used HDFS [19] as the Results Cache and MySQL Cluster [20] as the Repository for our implementation. MySQL Cluster is an in-memory database which provides high read/write throughput. It is a highly available, scalable and fault-tolerant store. We use ClusterJ to connect Pig with MySQL Cluster, because it is simple to use and can well satisfy our needs. We created one table in MySQL Cluster to represent the Repository, with four columns: hashcode - hash of the plan; filename - output path of the plan; frequency - the reuse frequency; last access - last time to access the record. Correspondingly, we created an annotated interface in Pig, having the same four properties. When a sub-plan is selected for materialization, we insert a new record at the MySQL Cluster table and store the result in HDFS. By using HDFS as the Results Cache, we can simply rewrite Pig Latin scripts by removing the corresponding sub-plan operators and adding a Load operator in their place, with the specified HDFS path. Similarly, in order to store a sub-plan, we only have to add a Store operator, after the chosen materialization point in the logical plan.

C. *Garbage Collection*

We have implemented garbage collection as a separate component, which can be invoked at will by the user.The Garbage Collection process includes three main steps: filter out the obsolete records, delete the corresponding outputs from the Results Cache and then delete the records from MySQL database. We have based the garbage collection policy on reuse frequency and the last access time. Since the frequently reused records tend to have a more recent access time than the less reused ones, our garbage collector is only based on the last access time threshold, which is a user-defined configuration parameter. For example, if we set the threshold to 5 days, the records that have not been accessed within 5 days from the time we ran the garbage collector will be deleted. In this case, the materialized results that have never been reused will also be eventually deleted, since their last access value does not exist. The reuse frequency threshold can also be specified by user, but we did not use it for our experiments.

VI. Evaluation

In this section, we describe the environment we used for evaluating our prototype, present and discuss our results.

A. *Environment Setup*

We have set up a Hadoop cluster and a MySQL cluster on top of OpenStack. We have used 20 Ubuntu 11.10 virtual machines, each one running Java(TM) SE Runtime Environment version 1.7.0. We have installed Hadoop version 1.0.4 and configured one Namenode and 15 Datanodes. The Namenode has 16 GB of RAM, 8 cores and 160GB disk, while the Datanodes have 4GB of RAM, 2 cores and 40 GB disk. For convenience, we have installed Pig 0.11 on the Hadoop Namenode. We have also used MySQL cluster version 7.2.12, with one manage node, one SQL node and 2 data nodes. Each node has 4GB of RAM, 2 cores and 40GB of disk space.

B. *Data and Queries*

We have used Jie Li's work [21] for running the TPC-H benchmark in Pig. We have generated the data sets using the DBGEN tools of the TPC-H Benchmark and have generated 107 GB of data in total. In order to create reuse opportunities, we have created some new queries by changing substitution parameters. Finally, we selected 20 queries in total for evaluation, 14 from the original TPC-H Benchmark queries and 6 newly created queries, with reuse opportunity.

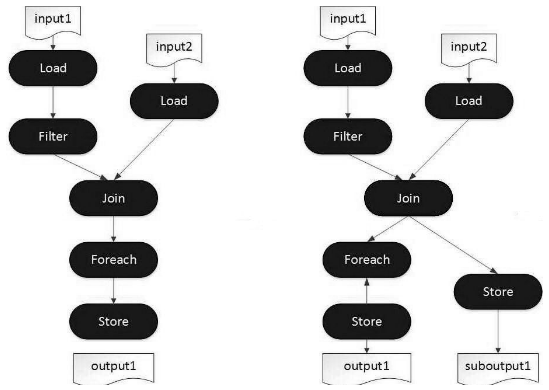

Fig. 4: Storing the Result of a Sub-Job

C. Sub-Plan Selection Strategy

We have configured m2r2 to materialize results after the Join and the CoGroup operators. For logical plan with multiple inputs, if one input branch changes, then the final Join of all the inputs must be recomputed. Thus, if the plan has multiple inputs, we only materialize the intermediate results on each input branch before the final Join operator. The TPC-H queries usually contain multiple inputs, such as Query 20. Starting from the Store operator, Match and Rewrite would not be performed until reaching the branches. An Example Pig logical plan showing a selection of a sub-plan to be materialized is shown in Figure 4.

D. Results

In this section, we present the results of our evaluation. We have first executed the set of the 20 TPC-H queries (14 original and 6 with modified substitution parameters) in Pig with and without enabling our materialization framework. We have run each test at least 5 times, ensuring that standard deviation of the measurements is under 8%. In the first set of experiments, we are interested in investigating the overhead that our materialization framework incurs compared to regular Pig scripts execution. Although we only discuss here the overhead regarding total execution time, we are also planning to look into storage overhead in future experiments. In the second set of experiments, we focus on quantifying the benefits gained by reusing materialized results for query execution. Finally, we also examine the performance of our implementation and the throughput of MySQL Cluster.

1) Materialization Overhead: Figure 5 shows a comparison of the execution time for the 14 original queries on Pig without materialization with the query execution time on Pig with materialization enabled. We clarify that, in this experiment, no materialized results were exploited, so that the measurements depict the pure overhead of the materialization mechanism. The numbers on x axis are in correspondence with the query numbers in TPC-H Benchmark. When having materialization enabled, the overhead consists of the Match and Rewrite algorithm, the optimization of the rewritten logical plan, storing execution statistics to MySQL Cluster and storing results in HDFS. We observe that for queries 4, 6, 12, 13 and 17 there is no overhead or negligible overhead. This is due to the fact that either no intermediate results were materialized or their size

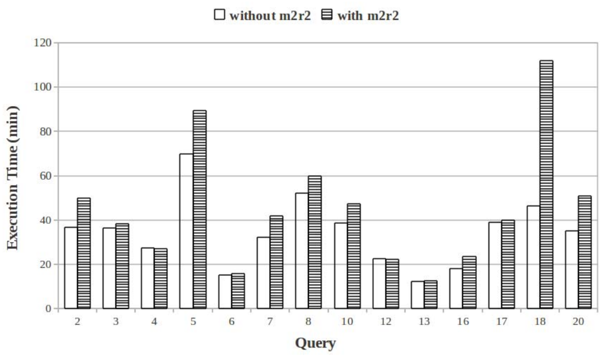

Fig. 5: Comparison of Query Execution Times on Pig without Materialization and Query Execution Times with Materialization. When the materialization mechanism is enabled, materialized results are produced but not exploited.

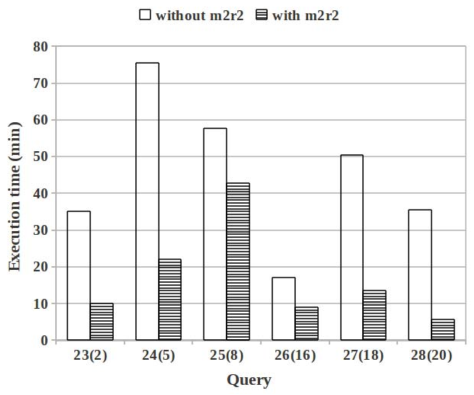

Fig. 6: Comparison of Query Execution Times With and Without Using Sub-Job Materialized Results

was too small. For queries 2, 3, 5, 7, 8, 10, 18 and 20 we notice very small overhead, from 5% to 25% over the total execution time. Finally, we observe a quite large overhead for query 18, around 120%. In this case, the size of the intermediate results chosen for materialization was very large. We expect this case not to be so frequent and note that this is a one-time overhead that will benefit all future matching queries.

2) Benefit of Reusing Plans and Sub-Plans: After having materialized the plans and selected sub-plans of the first 14 queries, we have run the 6 additional queries containing common sub-queries. The results are shown in Figure 6. The figure shows a comparison of query execution times of the system when using the reuse framework to exploit results of sub-jobs, with query execution times of the system without materialization. On the x axis, it is shown in parentheses the number of the original query from which each new script was produced. We observe a speedup of 50% - 80% for most queries, except for query 25, which has the lowest speed up of 30%. This is because we did not modify this particular query so that it can reuse its most I/O dominant sub-plan.

For the second part of this experiment, we run again all 20 queries, after having stored all relevant data in MySQL and HDFS, in order to measure the benefit of reusing results of whole jobs. As expected, the achieved speedup, shown in

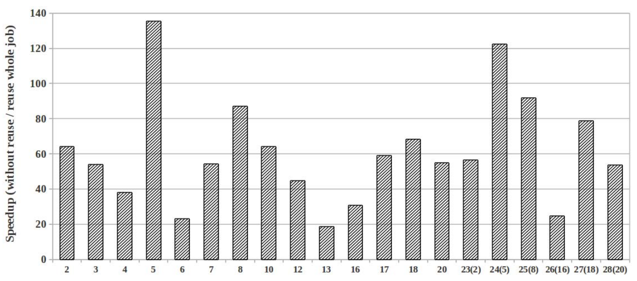

Fig. 7: Speedup When Using Results of Whole Jobs

Figure 7, is tremendous and the execution time is around 30 - 45 sec for all queries. The speedup depends on the base execution time, the size of the final results and the execution time variance. In this case, for all queries, the execution time essentially consists of the time to make a lookup in the repository and return the location of the stored results in HDFS. The total execution time of all 20 queries with materialization disabled is around 753 min and is reduced to around 13 minutes, when the results of whole jobs can be reused. This scenario is probably not realistic, but we believe it is highly indicative of the potential benefits of the materialization and reuse framework.

3) Non-I/O Overheads: The non-I/O overheads are introduced by the execution of the Match and Rewrite algorithm and the access to MySQL Cluster. We have found that the non-I/O overheads are negligible when compared to I/O overheads. Creating a session to access MySQL cluster takes the longest time. We have collected around 200 values of session creation time, 62.6% of them being around 780 ms. The execution time of the Match and Rewrite algorithm is also at the order of ms. The measured time includes computing the fingerprints, lookup of matched plan and MySQL Cluster read/write operations.

VII. Related Work

Restore [11] is a non-concurrent sharing framework built on top of Pig, which uses materialized results for query optimization. The main idea is very similar to our work but it is designed exclusively for Pig on Hadoop and operates on the physical level. Restore consists of four components: a sub-job enumerator, a matcher and rewriter, a repository and a sub-job selector. The sub-job enumerator selects sub-jobs for materialization and the matcher and rewriter performs match and rewrite of plans. In our implementation, both functionalities are performed by the Plan Matcher and Rewriter. The repository is used to store execution statistics and is analogous to MySQL Cluster in m2r2. An important implementation difference is that ReStore stores and matches the plan object, while we use fingerprints to identify and match plans. Moreover, garbage collection is not implemented. We have not included an comparative evaluation of our implementation with ReStore, because, to the best of our knowledge, the system was not publicly available at the time this paper was written.

Incoop [22] is a non-concurrent sharing framework that can reuse results for incremental computation. It detects input data changes by using content-based chunks instead of fixed-size chunks to locate files. During the map phase, results of the unchanged chunks are fetched from the file system and processed by mapper. During the incremental reduce phase, Incoop not only stores the final results, but also stores sub-computation results by introducing an additional phase, called the contraction phase. During contraction, the input of a reduce job is split into chunks which are processed by combiners. Thus, the result is computed by combining the results of unchanged chunks and the output of the combiners. Unlike Incoop, we focus at identifying the computations sharing the same input and not inputs sharing the same computations.

Microsoft has conducted remarkable research on query optimization by materializing results, DryadInc [6], the Nectar [7] and Comet [5], being the most representative systems related to our work. All are built upon the Dryad/DryadLINQ and like us, assume an append-only file system. DryadInc is similar to Incoop and focuses on non-concurrent sharing incremental computation. It uses two heuristics to do incremental computation: identical computation and mergeable computation. Nectar is a non-concurrent sharing framework, which aims to improve both computation efficiency and storage utilization. Regarding computation, it uses a program rewriter and a cache server to avoid redundant computations, similar to DryadInc. For data management, it uses a mechanism to store all executed programs and materialize all computation results. When the garbage collection process detects the existence of computation results that have not been accessed for a long time, it replaces them with the programs that produce them. Comet enables both concurrent sharing and non-concurrent sharing. It performs three kinds of optimizations: query normalization, logical optimization and physical optimization. In query normalization phase, a single query is split into sub-queries, which can be reused by other sub-queries. Moreover, each sub-query is tagged with a time stamp, so that the system can identify whether a set of sub-queries from different queries that can be executed together. During logical optimization, common sub-queries are detected and are executed for only once. Reordering is also enabled, in order to expose more common expressions. Comet also considers co-location when replicating materialized results, for reducing network traffic. During physical optimization, shared scan and shared shuffling are performed. While all the above systems assume Dryan/LINQ as the programming and execution environment, in our work, we have designed a language-independent and extensible framework for storing, managing and using previous job and sub-job results. Our initial prototype of the m2r2 framework is built on top of the Pig dataflow system and can be easily extended to support other platforms.

VIII. Conclusions and Future Work

The problem of computation redundancy is very relevant in big data systems. Several studies have shown large similarities in data analysis queries and suggest that any type of caching techniques would greatly benefit data analysis frameworks.

Following the idea of materialized views in relational databases, we have examined the possibility of porting this technique in big data environments. In this work, we present m2r2, a results materialization and reuse framework for high-level dataflow systems. We have examined and compared several popular high-level dataflow languages and execution engines and we have summarized their common properties and design characteristics. We observe that the majority of systems follow very similar designs in their upper layers, namely the language and logical plan layer, while their physical layers differ significantly. We, therefore, present a design for integrating a materialization and reuse framework after the logical layer of high-level dataflow processing systems.

We have implemented an initial prototype framework on top of Pig/Hadoop and have used the TPC-H Benchmark to evaluate our work. The results show that when there exists sharing opportunity, query execution time can be immensely reduced by reusing previous results. We also show that the induced overhead of materialization is quite small, around 25% of the total execution time without materialization, while non-I/O overheads are negligible. We note that both benefit and overhead are very sensitive to framework parameters, such as sub-job selection strategy and garbage collection policy, as well as specific workload characteristics.

There are a number of open issues and interesting paths to explore in the context of this research. First, we plan to extend m2r2, by decoupling it from the Pig framework and building a general, component-based and extensible framework, as described in Section IV. We then intend to integrate it with other popular high-level systems. We are also particularly interested in exploring possibilities of sharing and reusing results among different frameworks. Furthermore, we believe it is essential to obtain execution traces from industrial partners and big organizations, in order to evaluate the gains and overheads of the materialization framework under more realistic conditions. We would like to examine how benefits are related to different workload characteristics, data distributions and cluster sizes. We intend to minize the imposed overhead, by overlapping the materialization process with regular query execution, thus moving it out of the critical path of the execution. Finally, we also wish to examine the possibility of extending m2r2 in order to support incremental computations and exploit concurrent-sharing opportunities.

ACKNOWLEDGMENT

This work was supported in part by the Erasmus Mundus Joint Doctorate in Distributed Computing (EMJD-DC) funded by the Education, Audiovisual and Culture Executive Agency (EACEA) of the European Commission under the FPA 2012-0030, and in part by the End-to-End Clouds project funded by the Swedish Foundation for Strategic Research (SSF) under the contract RIT10-0043.

REFERENCES

[1] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava, "Building a high-level dataflow system on top of map-reduce: the pig experience," *Proc. VLDB Endow.*, vol. 2, no. 2, pp. 1414–1425, Aug. 2009.

[2] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: a warehousing solution over a map-reduce framework," *Proc. VLDB Endow.*, vol. 2, no. 2, pp. 1626–1629, Aug. 2009.

[3] K. S. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Y. Eltabakh, C.-C. Kanne, F. Özcan, and E. J. Shekita, "Jaql: A scripting language for large scale semistructured data analysis." *PVLDB*, vol. 4, no. 12, pp. 1272–1283, 2011.

[4] Y. Chen, S. Alspaugh, and R. Katz, "Interactive analytical processing in big data systems: a cross-industry study of mapreduce workloads," *Proc. VLDB Endow.*, vol. 5, no. 12, pp. 1802–1813, Aug. 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2367502.2367519

[5] B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, and L. Zhou, "Comet: batched stream processing for data intensive distributed computing," in *Proceedings of the 1st ACM symposium on Cloud computing*, ser. SoCC '10. New York, NY, USA: ACM, 2010, pp. 63–74.

[6] L. Popa, M. Budiu, Y. Yu, and M. Isard, "Dryadinc: reusing work in large-scale computations," in *Proceedings of the 2009 conference on Hot topics in cloud computing*, ser. HotCloud'09. Berkeley, CA, USA: USENIX Association, 2009.

[7] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang, "Nectar: automatic management of data and computation in datacenters," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–8.

[8] J. Goldstein and P.-A. Larson, "Optimizing queries using materialized views: a practical, scalable solution," in *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '01. New York, NY, USA: ACM, 2001, pp. 331–342.

[9] J. Yang, K. Karlapalem, and Q. Li, "Algorithms for materialized view design in data warehousing environment," in *Proceedings of the 23rd International Conference on Very Large Data Bases*, ser. VLDB '97. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997, pp. 136–145.

[10] S. Agrawal, S. Chaudhuri, and V. R. Narasayya, "Automated selection of materialized views and indexes in sql databases," in *Proceedings of the 26th International Conference on Very Large Data Bases*, ser. VLDB '00. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000.

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

[12] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica, "Shark: Sql and rich analytics at scale," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. New York, NY, USA: ACM, 2013, pp. 13–24.

[13] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10.

[14] V. Kalavri, V. Vlassov, and P. Brand, "Ponic: Using stratosphere to speed up pig analytics," in *Euro-Par*, 2013, pp. 279–290.

[15] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke, "Nephele/PACTs: A programming model and execution framework for web-scale analytical processing," in *Proceedings of the 1st ACM symposium on Cloud computing*, ser. SoCC '10. New York, NY, USA: ACM, 2010, pp. 119–130.

[16] A. Y. Halevy, "Answering queries using views: A survey," *The VLDB Journal*, vol. 10, no. 4, pp. 270–294, Dec. 2001.

[17] A. Gupta and I. S. Mumick, "Materialized views," A. Gupta and I. S. Mumick, Eds. Cambridge, MA, USA: MIT Press, 1999, ch. Maintenance of materialized views: problems, techniques, and applications, pp. 145–157.

[18] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey, "Dryadlinq: a system for general-purpose distributed data-parallel computing using a high-level language," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 1–14.

[19] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, ser. MSST '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10.

[20] "MySQL Cluster," http://www.mysql.com/products/cluster, [Online; Last accessed Aug 2013].

[21] "TPC-H Benchmark on Pig," http://issues.apache.org/jira/browse/PIG-2397, [Online; Last accessed Aug 2013].

[22] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin, "Incoop: Mapreduce for incremental computations," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*, ser. SOCC '11. New York, NY, USA: ACM, 2011, pp. 7:1–7:14.