

HDM: Optimized Big Data Processing with Data Provenance

Dongyao Wu ^{†,‡}, Sherif Sakr ^{†,‡,*}, Liming Zhu ^{†,‡}

[†]Data61, CSIRO, Sydney, Australia

[‡]School of Computer Science and Engineering, University of New South Wales, Sydney, Australia

^{*}King Saud bin Abdulaziz University for Health Sciences, National Guard, Riyadh, Saudi Arabia

{firstname.lastname}@data61.csiro.au

ABSTRACT

Big Data applications are becoming more complex and experiencing frequent changes and updates. In practice, manual optimization of complex big data jobs is time-consuming and error-prone. Maintenance and management of evolving big data applications is a challenging task as well. We demonstrate HDM, Hierarchically Distributed Data Matrix, as a big data processing framework with built-in data flow optimizations and integrated maintenance of data provenance information that supports the management of continuously evolving big data applications. In HDM, the data flow of jobs are automatically optimized based on the functional DAG representation to improve the performance during execution. Additionally, comprehensive meta-data related to explanation, execution and dependency updates of HDM applications are stored and maintained in order to facilitate the debugging, monitoring, tracing and reproducing of HDM jobs and programs.

Keywords

Big Data; Data Flow Optimization; Provenance Management

1. INTRODUCTION

We are experiencing the era of big data that has been fuelled by the striking speed of the growth in the amount of data that has been generated and consumed. Several big data processing frameworks (e.g., MapReduce [2], Spark [6] and Flink [1], etc.) have been introduced to deal with the challenges of processing the ever larger data sets [3]. These frameworks significantly reduce the complexity of writing large scale data-oriented applications. However, in practice, as big data programs and applications have become more and more complicated, it is almost impossible to manually optimize the performance of programs written by diversified programmers. Therefore, built-in optimizers are crucial for tackling the challenges of improving the performance of executing those hand-written programs and applications. At

the same time, realistic data analytics applications are continuously evolving in order to deal with the non-stop changes in the real world. In practice, managing and analyzing those continuously evolving big data applications have resulted in big technical debts [4]. Therefore, there are increasing requirements for data provenance to support analyzing, tracing and reproduction of historical versions of data analytics applications.

In this paper, we demonstrate HDM, (Hierarchically Distributed Matrix) [5], a big data processing framework with built-in data optimizations for execution and data provenance supports for managing continuously evolving big data applications. In particular, HDM is a lightweight, functional and strongly-typed data representation which contains complete information (such as data format, locations, dependencies and functions between input and output) to support parallel execution of data-driven applications [5]. Exploiting the functional nature of HDM enables deployed applications of HDM to be natively integrable and reusable by other programs and applications. In addition, by analyzing the execution graph and functional semantics of HDMs, multiple optimizations are provided to automatically improve the execution performance of HDM data flows. Moreover, by drawing on the comprehensive information maintained by HDM graphs, the runtime execution engine of HDM is also able to provide provenance and history management for submitted applications.

2. HDM FRAMEWORK

2.1 System Overview

Fig 1 shows the system architecture of the HDM runtime engine which is composed of three main components:

- *Runtime Engine*: is responsible for the management of HDM jobs such as explaining, optimization, scheduling and execution. Within the runtime engine, the AppManager manages the information of all deployed jobs. TaskManager maintains the activated tasks for runtime scheduling in the Schedulers; Planner and Optimizers interpret and optimize the execution plan of HDMs in the explanation phases; HDM manager manages the information and states of the HDM blocks in the entire cluster; Execution Context is an abstraction component to support the execution of scheduled tasks on either local or remote nodes.
- *Coordination Service*: is composed of three types of coordinations: cluster coordination, block coordination

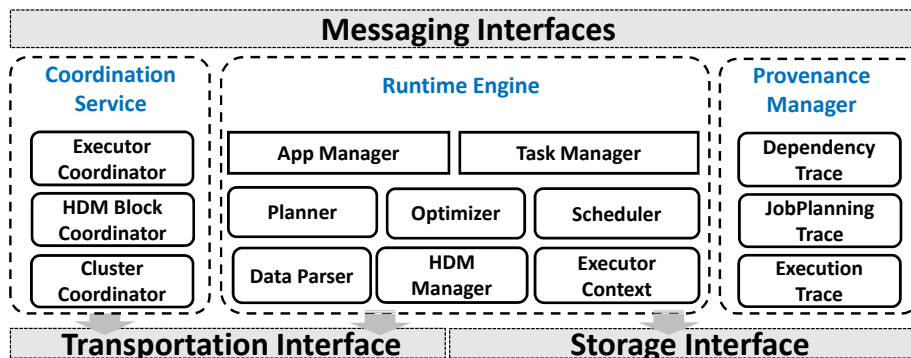


Figure 1: System Architecture of HDM Framework.

and executor coordination. They are responsible for the coordination and management of node resources, distributed HDM data blocks and executors on workers, respectively.

- *Data Provenance Manager*: is responsible to interact with the HDM runtime engine to collect and maintain data provenance information (such as DependencyTrace, JobPlanningTrace and ExecutionTrace) for HDM applications. Those information can be queried and obtained by client programs through messages for the usage of analysis or tracing.

2.2 HDM Data Flow Optimization

One key feature of HDM is that, the execution engine contains built-in planners and optimizers to automatically optimize the functional data flow of submitted applications and jobs. During explanation of HDM applications, the data flow are represented as DAGs with functional dependencies among operations. The HDM optimizers traverse through the DAG to reconstruct and modify the operations based on optimization rules to obtain more optimal execution plans. Currently, the optimization rules implemented in the HDM optimizers include: function fusion, local aggregation, operation reordering and data caching for iterative jobs [5].

- *Function fusion*. During optimization, the HDM planner combines the lined-up non-shuffle operations into one operation with high-order function so that the sequence of operations can be compute within one task rather than separate ones to reduce redundant intermediate results and task scheduling. This rule can be applied recursively on a sequence of fusible operations to form a compact combined operation.
- *Local Aggregation*. Shuffle operations are very expensive in the execution of data-intensive applications. If a shuffle operation is followed with some aggregations, in some cases, the aggregation or part of the aggregation can be applied before the shuffling stage. During optimization, HDM planner tries to move those aggregation operations forward before the shuffling stage to reduce the amount of data that needs to be transferred during shuffling.
- *Operation reordering/reconstruction*. Apart from aggregations, there are a group of operations which filter out a subset of the input during execution. Those

operations are called pruning operations. The HDM planner attempts to lift the priority of the pruning operations while sinking the priority of shuffle-intensive operations to reduce the data size that needs to be computed and transferred across the network.

- *Data Caching*. For many complicated and pipelined analytics jobs (such as machine learning algorithms), some intermediate results of the job could be reused multiple times by the subsequent operations. Therefore, it is necessary to cache those repetitively used data to avoid redundant computation and communication. In this case, HDM planner counts the reference for the output of each operation in the functional DAG to detect the potential points that intermediate results should be cached for reusing by subsequent operations.

During optimization process, the rule above are applied one by one to reconstruct the HDM DAG and the optimization can last multiple iterations until there is no change in the DAG or it has reached the maximum number of iterations. The HDM optimizer is also designed to be extendable by adding new optimization rules by developers when it is needed.

2.3 Data Provenance Supports in HDM

It is normally tedious and complicated to maintain and manage applications that are continuously evolving and being updated. In HDM, drawing on comprehensive meta-data information maintained by HDM models, the runtime engine is able to provide data provenance supports including execution tracing, version control and job replay in the dependency and execution history management component.

Basically, the HDM server maintains three types of meta-data about each submitted HDM jobs including ExecutionTrace, JobPlanningTrace and DependencyTrace.

- *DependencyTrace*. For every submitted HDM program, the server stores and maintains the dependent libraries required for execution. The dependencies and update history are maintained as a tree structure. Based on this information, users are able to reproduce any version of the submitted applications in the history.
- *JobPlanningTrace*. The HDM server also stores the explanation and planning traces for every HDM applications. JobPlanningTrace includes the logical plan, optimizations applied and final physical execution plan after being parallelized.

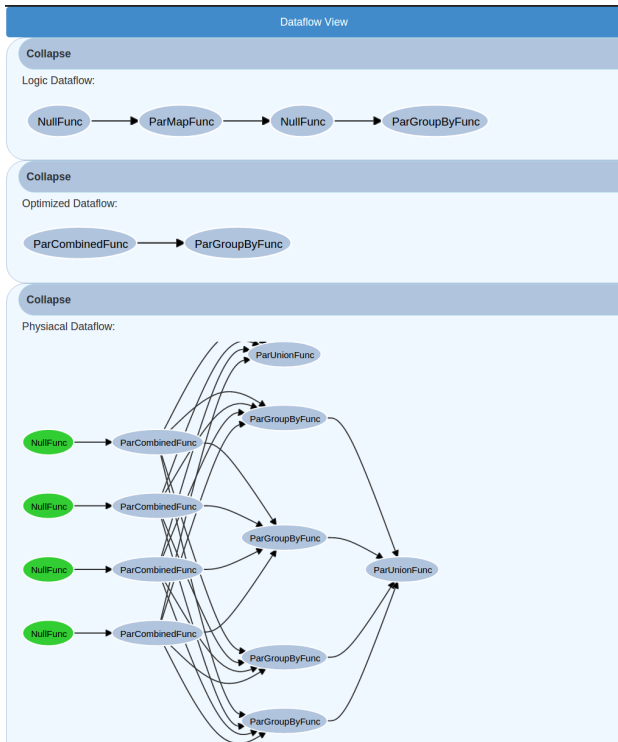


Figure 2: Dataflow Visualization of HDM Applications.

- *ExecutionTrace.* During execution, the HDM server also maintains all the runtime information (execution location, input/output, timestamps and execution status, etc.) related to each executed task and job. These information are very meaningful to monitor and trace back the process of execution of historical jobs and applications.

Drawing on the three types of information maintained in the HDM server, client-side programs can send messages to query and obtain the history and provenance information, so that users and administrators can profile, debug and apply analysis to the deployed applications throughout their life cycles.

3. DEMONSTRATION SCENARIOS

In this demonstration, we will present to the audience the HDM framework¹ from four main aspects: cluster resource monitoring, visualisation data flow optimization, execution history tracing, version-control and dependency management. The demonstration will be conducted on AWS EC2 with one M3.Large instance as the master and 10 nodes M3.XLarge instances as the workers.

To show how HDM optimizes the data flow and provides data provenance support for its applications, we will present an example of Twitter analysis scenario that consists of the following two Tweets analysis programs²:

- The first program, presented in Listing 1, looks for the

¹The source code of the HDM framework is available on <https://github.com/dwu-csiro/HDM>

²A demonstration screencast is available on <https://youtu.be/Gsz7z5bQ1zI>

Tweets that are related to recent election events by checking the hashtag of the input Tweets.

- The second program, presented in Listing 2, finds out the Tweets that are related to two candidates: “Trump” and “Hillary” and count the amount for each of them.

Listing 1: Code Snippet of Finding out Tweets

```
val input = HDM("hdfs://10.10.0.100:9091/user/tweets")
val tweets = input.map{ line =>
    val seq = line.split(",")
    Tweet(seq)
}
val grouped = tweets.groupBy(t => t.hashTag)
val results = grouped.findByKey(_.contains("election"))
```

Listing 2: Code Snippet of Hashtag Counting for Interested Tweets

```
val input = HDM("hdfs://10.10.0.100:9091/user/tweets")
val tweets = input.map{ line =>
    val seq = line.split(",")
    Tweet(seq)
}
val grouped = tweets.groupBy(t => t.hashTag)
val trumpN = grouped.findByKey(_ == "Trump").count
val hillaryN = grouped.findByKey(_ == "Hillary").count
println(trumpN / hillaryN)
```

Cluster Resource Management. In the first part of the demo, we will show the cluster resource monitor of the HDM manager. The HDM server maintains the resource-related information of all the workers within the cluster. In the HDMConsole, it is able to monitor the resource utilization information (such as CPU, Memory, Network and JVM) for each worker in real time. Therefore, cluster administrator is able to use these information and easily supervise and understand the status of every worker as well as the entire cluster.

Dataflow Optimizations. The second part of the demo shows how the Tweets programs are represented in the HDM DAG and how it is explained, optimized and parallelized by the planner.

- For the first program, the HDM optimizer applies operations reordering to lift the pruning operation `findByKey` to be in front of the shuffle operation `groupBy`. Then the optimizer applies function fusion rule to combine `map` and `findByKey` into a single composite operation.
- For the second program, the HDM optimizer applies operation reordering to move the `findByKey` operation to be in front of `groupBy` then applies local aggregation `count` by adding local count in front of `groupBy`. Lastly, it detects the input tweets that are reused by two operations so that the optimizer can add a cache point after the compute operation that generates the output of `tweets`.

The HDM server maintains all the related meta-data (such as the creator, original program, logical plan, physical plan, etc.) to all the submitted HDM applications. In the demonstration, the HDM console visualizes the original logical flow, optimized logical flow and parallelized physical graph

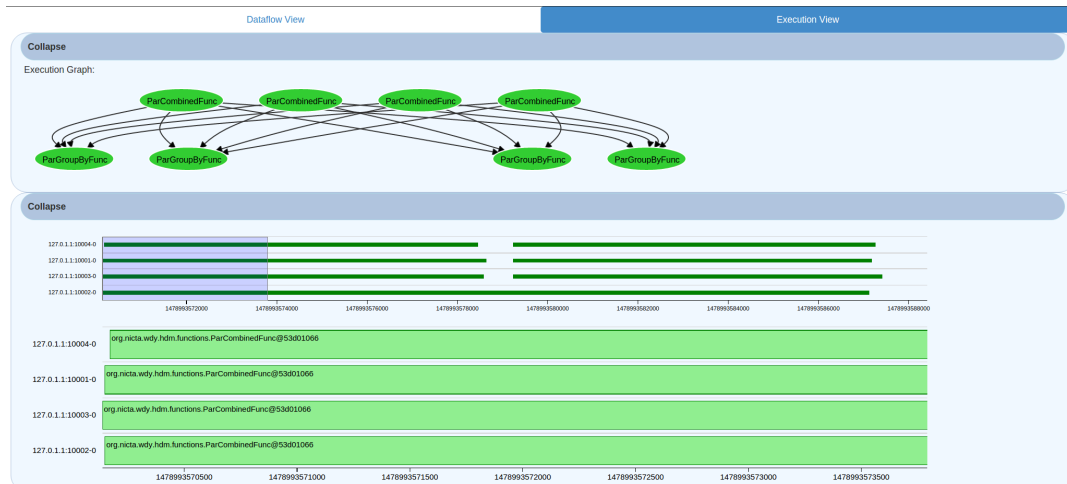


Figure 3: Execution Traces of HDM Applications.

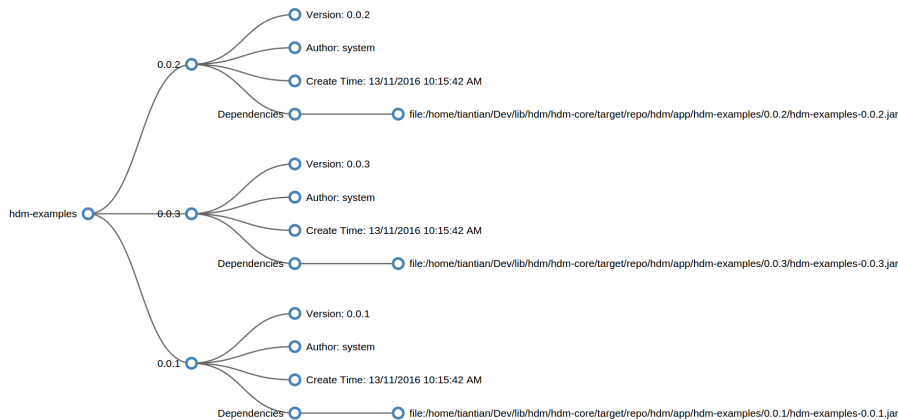


Figure 4: Dependency Management and Version Control of HDM.

for each execution instances of the HDM applications (Figure 2).

Execution History Tracing. In the third part of the demo, we will show how the execution process can be tracked during and after execution. The HDM server collects and stores the runtime information for each execution task and structures them into DAG based on the task dependencies. During or after the execution of the tasks, the HDM server also updates the status in the stored meta-data when it has received the notification messages. The HDM console also summarizes those information and presented it into a view of execution lanes for each core of the workers (Figure 3).

Dependency Management and Version Control. In the last part of the demo, we will show how the HDM server manages the dependencies and provides version control for submitted applications. The dependency and history manager stores all the updating history of each HDM applications and organizes them into a tree based structure. As a result, administrator users are able to query, analyze and reproduce the historical HDM applications using those dependencies information (Figure 4).

Besides the framework demonstration, we will also discuss in more details about the design choices that we have made on defining the different components of the framework. In addition, performance comparison with the Spark frame-

work [6], using the example scenario, will be presented to demonstrate the efficiency of the HDM optimization techniques.

4. REFERENCES

- [1] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flinkTM: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.
- [2] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1), 2008.
- [3] S. Sakr. *Big Data 2.0 Processing Systems - A Survey*. Springer Briefs in Computer Science. Springer, 2016.
- [4] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, and M. Young. Machine learning: The high interest credit card of technical debt. In *SE4ML: Software Engineering for Machine Learning*, 2014.
- [5] D. Wu, S. Sakr, L. Zhu, and Q. Lu. Composable and Efficient Functional Big Data Processing Framework. In *IEEE Big Data*, 2015.
- [6] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *HotCloud*, 2010.