This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2017.2690906, IEEE Transactions on Big Data

HDM: A Composable Framework for Big Data Processing

Dongyao Wu*†, Sherif Sakr†‡, Liming Zhu*†, Qinghua Lu*§
*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
§College of Computer and Communication Engineering, China University of Petroleum, Qingdao, China

Abstract—Over the past years, frameworks such as MapReduce and Spark have been introduced to ease the task of developing big data programs and applications. However, the jobs in these frameworks are roughly defined and packaged as executable jars without any functionality being exposed or described. This means that deployed jobs are not natively composable and reusable for subsequent development. Besides, it also hampers the ability for applying optimizations on the data flow of job sequences and pipelines. In this paper, we present the Hierarchically Distributed Data Matrix (HDM) which is a functional, strongly-typed data representation for writing composable big data applications. Along with HDM, a runtime framework is provided to support the execution, integration and management of HDM applications on distributed infrastructures. Based on the functional data dependency graph of HDM, multiple optimizations are applied to improve the performance of executing HDM jobs. The experimental results show that our optimizations can achieve improvements between 10% to 40% of the Job-Completion-Time for different types of applications when compared with the current state of art, Apache Spark.

Index Terms—big data processing, parallel programming, functional programming, distributed systems, system architecture.

1 Introduction

Big Data has become a popular term which is used to describe the exponential growth and availability of data. The growing demand for large-scale data processing and data analysis applications spurred the development of novel solutions to tackle this challenge [10]. For about a decade, the MapReduce framework has represented the defacto standard of big data technologies and has been widely utilized as a popular mechanism to harness the power of large clusters of computers. In general, the fundamental principle of the MapReduce framework is to move analysis to the data, rather than moving the data to a system that can analyze it. It allows programmers to think in a data-centric fashion where they can focus on applying transformations to sets of data records while the details of distributed execution and fault tolerance are transparently managed by the framework. However, in recent years, with the increasing applications' requirements in the data analytics domain, various limitations of the Hadoop framework have been recognized and thus we have witnessed an unprecedented interest to tackle these challenges with new solutions which constituted a new wave of mostly domain-specific, optimized big data processing platforms [11].

In recent years, several frameworks (e.g. Spark [16], Flink, Pregel [7], Storm) have been presented to tackle the ever larger datasets on using distributed clusters of commodity machines. These frameworks significantly reduce the complexity of developing big data programs and applications. However, in reality, many real-world scenarios require pipelining and integration of multiple big data jobs. There are more challenges when applying big data technology in practice. For example, consider a typical online machine learning pipeline as shown in Fig. 1, the pipeline consists of three main parts (Fig. 1): the data parser/cleaner, feature extractor and classification trainer. In the pipeline, components like feature extractor and classification trainer are normally commonly-used algorithms for many machine learning applications. However, in current big data platform such as MapReduce and Spark, there is no proper way to share and expose a deployed and well-tuned online component to other developers. Therefore, there are massive and even unseen redundant development in big data applications. In addition, as the pipeline evolves, each of the online components might be updated and re-developed, new components can also be added in the pipeline. As a result, it is very hard to track and check the effects during the evolving process. Google's recent report [12] shows the challenges and problems that they have encountered in managing and evolving large scale data analytic applications. Furthermore, as the pipeline become more and more complicated, it is almost impossible to manually optimize the performance for each component not mentioning the whole pipeline. To address the auto-optimization problem, Tez [9] and FlumeJava [3] were introduced to optimize the DAG of MapReduce-based jobs while Spark relies on Catalyst to optimize the execution plan of SparkSQL [2].

To sum up, the main challenges for current complicated analytic applications can be listed below:

- Many real-world applications require a chain of operations or even a pipeline of data processing programs.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2017.2690906, IEEE Transactions on Big Data

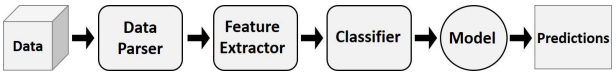


Figure 1. A simple image classification pipeline.

Optimizing a complicated job is difficult and optimizing pipelined ones are even harder. Additionally, manual optimizations are time-consuming and error-prone and it is almost impossible to manually optimize every program.

- Integration, composition and interaction with big data programs/jobs are not natively supported: Many practical data analytics and machine learning algorithms require combination of multiple processing components each of which is responsible for a certain analytical functionality. A key limitation for existing frameworks such as MapReduce and Spark is that jobs are roughly defined and packaged as binary jars and executed as black-boxes without exposing any information about the functionalities. As a result of this, deployed jobs are not natively composable and reusable for subsequent development and integration.
- Maintenance and management of evolving big data applications are complex and tedious. In a realistic data analytic process, data scientists need to explore the datasets and tune the algorithms back and force to find out a more optimal solution. Mechanisms such as history tracking and reproducibility of old-version programs are of great significance for helping data scientist not be lost during exploring and evolving their data analytic programs.

In order to tackle the above challenges, we believe that by improving the basic data and task models, these problems could be addressed to a great extent at the big data execution engine level. In particular, we present the Hierarchically Distributed Data Matrix (HDM) [14] along with the system implementation to support the writing and execution of composable and integrable big data applications. HDM is a light-weight, functional and strongly-typed meta-data abstraction which contains complete information (such as data format, locations, dependencies and functions between input and output) to support parallel execution of data-driven applications. 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. In particular, the main contributions of this paper can be summarized as follows:

- HDM, a lightweight, functional, strongly-typed data abstraction for developing and describing data-parallel applications.
- Based on the functional data dependency graph, optimizations include function fusion, local aggregation, operation reordering and caching are introduced to improve the performance of HDM jobs.
- A HDM system implementation provisioning that supports the execution, integration, history and dependency management of HDMs applications on distributed environments.
- Comprehensive benchmarks including: basic primitives, pipelined operations, SQL queries and iterative jobs (ML algorithms) are tested to evaluate the performance of HDM compared with the current state-of-art big data processing framework - Apache Spark.

TABLE 1. ATTRIBUTES OF HDM

Attribute	Description
ID	The identifier of a HDM. It must be unique within each HDM context.
inType	The input data type of computing this HDM.
outType	The output data type of computing this HDM.
category	The node type of this HDM. It refers to either DFM or DDM.
children	The source HDMs of a HDM. It describes from where this HDM can be computed.
distribution	The distribution relation of children blocks, including horizontal and vertical.
dependency	The data dependency for computing this HDM from its children. There are four types of data dependencies as 1:1, 1:N, N:1, N:N.
function	The function applied on input to calculate the output. This function can be a composed one and must have the same input and output type as this HDM.
blocks	The data blocks of this HDM. For DFM it can be an array of IDs for children DDM; This field is only available after all children of this HDM are computed.
location	It refers to the URL address of this HDM on local or remote nodes. For DDM, the actual data are loaded according to the protocol in the URL such as hdfs, file, mysql and hdm.
state	Current state of this HDM. A HDM can exist in different phases such as Declared, Computed and Removed.

The remainder of this paper are organized as follows. Section 2 introduces the representation and basic features of HDM. Section 3 describes the programming model of HDM. Section 4 presents the major optimizations applied to the HDM data flow. Section 5 presents the system architecture and implementations of the HDM runtime system. Section 6 describes the dependency and history management of HDM applications. In Section 7 presents a case study of writing pipelined applications in HDM and compares the performance of HDM with Apache Spark. In Section 8, we discuss the related work before we conclude the paper in Section 9.

2 REPRESENTATION OF HDM

Programming abstraction is the core of our framework, therefore, we first introduce our Hierarchically Distributed Data Matrix (HDM) which is a functional, strongly-typed meta-data abstraction for writing data-parallel programs.

2.1 Attributes of HDM

Basically, a HDM is represented as *HDM[T, R]*, in which *T* and *R* are data types for input and output, respectively. The HDM itself represents the function that transforms data from input to output. Apart from these core attributes, HDM also contains information like data dependencies,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2017.2690906, IEEE Transactions on Big Data

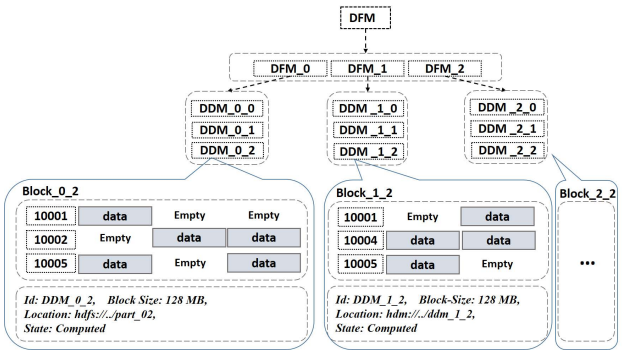


Figure 2. Data Model of HDM.

location, distribution to support optimization and execution. The attributes of a HDM are listed in TABLE I. Those attributes are chosen based on the design requirements of HDM. *'inType'* and *'outType'* is used to guarantee the type correctness during optimization and composition. *'category'* is used to differentiate DFM and DDM during job planing, optimization and execution. *'children'* and *'dependency'* are used to reconstruct the HDM DAG during job planing and optimization. The attribute *'function'* is the core function of the HDM, it illustrates the how to compute the output of this HDM. Attribute *'blocks'* are used to specify the location of output for this HDM and it can be used as the input of a subsequent HDM computation. *'location'* represents that in which context or where the HDM is declared and managed. *'state'* is used to manage and check the runtime status of HDMs. Based on these attributes above, HDM supports the following basic features:

- *Functional*: A HDM is essentially a structured representation of a function that computes the output from some input. The computation of a HDM is focused on the evaluation of the contained function on the input data set (as children in HDM). During the computation of a HDM, no side effects are involved.
- *Strongly-typed*: HDM contains at least two explicit data types, the input type and output type, which are derived from the formats of the input and output based on the enclosed function. Note that strongly-typed in HDM means type information is explicitly included in HDM job interpreting, optimization and composition to guarantee the compatibility of data types.
- *Portable*: A HDM is an independent object that contains complete information for a computation task. Therefore, a HDM task is portable and can be moved to any nodes within the HDM context for execution.
- *Location-aware*: HDMs contains the information of the location (represented as formatted URL) of the inputs and outputs. Although, some location information is only available during runtime, it facilitates the ability to apply some optimizations for data localizations during the planning phases.

2.2 Categorization of HDM

As shown in Fig. 2, HDM is a tree-based structure which consists of the following two types of nodes:

- *Distributed Data Matrix* (DDM): The leaf-nodes in a HDM hierarchy hold the references of actual data and are responsible for performing atomic operations on data blocks. A DDM maintains the actual information such as: ID, size, location and status of one data block.
- *Distributed Functional Matrix* (DFM): The non-leaf nodes hold both the operational and distribution of relations for children HDMs; DFMs hold the specific functions about how to compute its output from the children HDM(can be either DFM or DDM). During execution, it is also responsible for collecting and aggregating the results from children nodes when necessary.

From the functional perspective, a DDM can be considered as a function which maps a path to an actual data set. Essentially, a DDM can be represented as *HDM [Path, T]*. During execution, data parsers are wrapped to load data from the data path according to their protocols and then transform the input to the expected outgoing formats of the DDM. A DFM is considered as a higher-level representation which focuses on the functional dependency for HDMs to serve the planning phases. Before execution, DFMs will be further explained as DDM dependencies according to data locations and the expected parallelism.

The separation of DFM and DDM provides different levels of views to support different levels of planning and optimization. In addition, the hierarchy of DFM and DDM also ensures that the local computation on data node is not concerned about data movement and coordination between siblings, thus leaving the parent nodes free to apply the aggregation steps.

2.3 Data Dependencies of HDM

In principle, data dependencies between HDMs affect when and how to compute HDMs from their children or predecessors. In particular, by performing operations on HDM, data dependencies are implicitly added between pre and post HDM nodes in the data flow. Basically, there are four types of dependencies in HDM (as shown in Fig. 3):

- *One-To-One* (1:1): one partition of input is only used to compute one partition of the output; Therefore, different partitions of the HDM can be executed in parallel without any intercommunication. Operations such as *Map*, *Filter*, *Find* would introduce a *One-To-One* dependency in the dataflow.
- *One-To-N* (1:N): one partition of input is used to compute multiple partitions of the output while one output partition only requires the input from one partition; Depending on the partition function, a *Partition/Repartition* operations would introduce a *One-To-N* dependency in the dataflow.
- *N-To-One* (N:1): one partition of input is only used to compute one partition of the output while one output partition requires multiple input partitions; Operations such as *Join*, *Reduce*, *ReduceByKey* would introduce a *N-To-One* dependency in the dataflow.
- *N-To-N* (N:N): Any other dependencies are considered as *N-To-N* dependency where one partition of input are used to compute multiple output partitions while one output partition also requires multiple input partitions. *GroupBy*, *CoGroup* and some specific *Partition* operation introduces *N-to-N* dependencies to the dataflow.

In practice, data dependency information represent a crucial aspect during both execution and optimization in

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2017.2690906, IEEE Transactions on Big Data

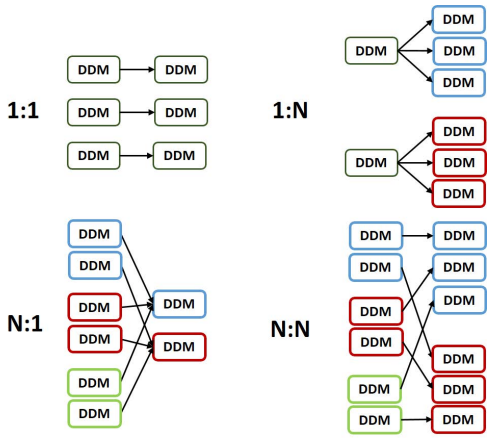


Figure 3. Data Dependencies of HDM.

TABLE 2. SEMANTICS OF BASIC FUNCTIONS

Function	Semantics
Null	Do nothing but return the input.
Map $f : T \rightarrow R$	F_p: $List[T].map(f)$ F_a: $List_1[R]$ += $List_2[T].map(f)$ F_c: $List_1[R]$ + $List_2[R]$
GroupBy $f : T \rightarrow K$	F_p: $List[T].groupBy(f)$ F_a: $List_1[K,T] \bigcup List_2[T].groupBy(f)$ F_c: $List_1[K,T] \bigcup List_2[K,T]$
Reduce $f : (T,T) \rightarrow T$	F_p: $List[T].reduce(f)$ F_a: $List_2[T].foldBy(zero = T_1)(f)$ F_c: $f(T_1, T_2)$
Filter $f : T \rightarrow Bool$	F_p: $List[T].filterBy(f)$ F_a: $List_1[T]$ += $List_2[T].filterBy(f)$ F_c: $List_1[T]$ + $List_2[T]$

order to decide how a HDM is computed, and which optimizations can be applied on the data flow, if at all.

3 PROGRAMMING ON HDM

One major target of contemporary big data processing frameworks is to ease the complexity for developing data-parallel programs and applications. In HDM, functions and operations are defined separately to balance between performance and programming flexibility.

3.1 HDM Functions

In HDM, a function specifies how input data are transformed as the output. Functions in HDM have different semantics targeting different execution context. Basically, one HDM function may have three possible semantics, indicated as F_p, F_a, F_c:

$$F_p : List[T] \rightarrow List[R] \quad (1)$$
$$F_a : (List[T], List[R]) \rightarrow List[R] \quad (2)$$
$$F_c : (List[R], List[R]) \rightarrow List[R] \quad (3)$$

F_p is the basic semantics of a function which specifies how to process one data block. The basic semantics of HDM function assume that the input data is organized as a sequence of records with type T. Similarly, the output of all the functions are also considered as a sequence of records. Based on the type compatibility, multiple functions can be directly pipelined.

F_a is the aggregation semantics of a function which specifies how to incrementally aggregate a new input partition to the existing results of this function. Functions are required to be performed on multiple data partitions when the input is too large to be fit into one task. The aggregation semantics are very useful under the situations in which accumulative processing could get better performance. Aggregation semantics exist for a function only when it is capable to be represented and calculated in an accumulative manner.

F_c is the combination semantics for merging multiple intermediate results from a series of sub-functions to obtain the final global output. It is also a complement for the aggregation semantics when a function is decomposable using the divide-combine pattern.

During the explanation of HDM jobs, different semantics are automatically chosen by planers to hide users from functional level optimizations. An illustration of the semantics of some basic HDM functions are listed in TABLE 2. During programming, operations in HDM are exposed as functional interfaces for users to use. Due to the declarative and intense manner offered by functional interfaces, users are able to write a `WordCount` program in HDM as shown in Fig. 4.

```
wordcount = HDM.string("path1","path2").map(_.split(","))
            .flatMap( w ⇒ (w, 1)
            .groupBy(t ⇒ t._1).reduceByKey(_ + _)
```

Figure 4. Example of writing a word-count program in HDM

3.2 HDM Composition

In functional composition, one function $f : X \rightarrow Y$ can be composed with another function $g : Y \rightarrow Z$ to produce a high-order function $h : X \rightarrow Z$ which maps X to $g(f(X))$ in Z. HDM inherits the idea of functional composition to support two basic types of composition:

$$HDM[T,R] \quad compose \quad HDM[I,T] \Rightarrow HDM[I,R] \quad (4)$$
$$HDM[T,R] \quad andThen \quad HDM[R,U] \Rightarrow HDM[T,U] \quad (5)$$

- *compose*: A HDM with input type T and output type R can accept a HDM with input type I and output type T as an input HDM to produce a HDM with input I and output R.
- *andThen*: A HDM with input type T and output type R can be followed by a HDM with input type any R and output type U as the post-operation to produce a new HDM with input T and output U.

These two patterns are commonly used in functional programming and can be recursively used in HDM sequences to achieve complicated composition requirements. In our system, composition operations are implemented as the basic primitives for HDM compositions and data flow optimizations.

3.3 Interaction with HDM

TABLE 3. ACTIONS AND RESPONSES FOR INTEGRATION

Action	Response
compute	References of computed HDMs.
sample	Iterator of a sampled sub-set for computed records.
count	Length of computed results.
traverse	Iterator of all computed records.
trace	Iterator of task information for last execution.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2017.2690906, IEEE Transactions on Big Data

HDM applications are designed to be interactive during runtime in an asynchronous manner. In particular, HDM programs can be written and embedded into other programs as normal code segments. Then, by triggering the action interfaces (listed in TABLE 3), jobs are dynamically submitted to the related execution context which can be either a multi-core threading pool or a cluster of workers. Fig. 5 shows how to interact with the `WordCount` job and print out the output on the client side.

```
wordcount.traverse(context = "10.10.0.100:8999")
onComplete {
    case Success(resp) ⇒ resp.foreach(println)
    case Failure(exception) ⇒ println(exception)}
```

Figure 5. Print output of HDM on the client side

4 Data Flow Optimizations on HDM

During runtime, HDM jobs are represented as functional DAG graphs, on which multiple optimizations could be applied to get better performance. Before execution, HDM optimizer traverses the DAG of HDM in a deep-first manner as shown in Fig. 6. During traversing, the optimizer checks each node scope (each scope contains the information of: current node, parent and its children and their data dependencies and functions) matches any of the optimization rules then reconstruct the scope based on the matched rule.

In the current implementation of HDM optimizer, there are four basic optimization rules: Local Aggregation, Re-ordering, Function Fusion and HDM Caching. In the remaining of this section, we will take the `WordCount` program (Fig. 4) as an example to explain each optimization rule and how they are applied to a HDM job.

4.1 Local Aggregation

Local aggregation is a very useful approach to reduce the communication cost for shuffle operations with aggregation semantics. For this kind of operations (such as *ReduceBy* and *FoldBy*), an aggregation operation can be applied before the shuffle phase, so that the amount of data can be significantly reduced for shuffling. Then, in the following step, a global aggregation is performed to compute the final results. The local aggregation rule in HDM can be specified as:

Local Aggregation Rule: Given a $HDM[T, R]$ with function $f : T \rightarrow R$, if HDM has *N-to-One* or *N-to-N* dependency and f has the semantics of aggregation then HDM can be split as multiple parallel $HDM_p[T, R]$ (with function f and one-to-one dependency) that is followed by $HDM_g[R, R]$ with the aggregation semantics F_a and original shuffle dependency.

For the example of `WordCount` program (in Fig. 4), the data flow of the job is initially explained as Fig. 7(a). By detecting the aggregation operation *ReduceByKey* after *GroupBy*, parallel branches are added to aggregate the data before the shuffling step. During the shuffle reading phase, the aggregate semantics of *ReduceByKey* function is applied to get the correct results from two sub-aggregating functions. The optimized data flow is shown in Fig. 7(b).

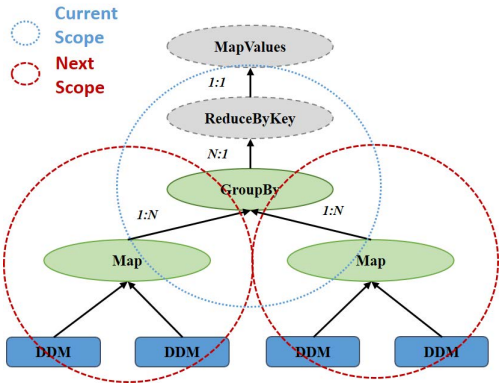


Figure 6. Traversing on the Data flow of HDM during Optimization.

4.2 Function Fusion

In HDM, we define fusible HDMs as a sequence of HDMs that start with *One-To-One* or *N-To-One* data dependency and end with *One-To-One* or *One-To-N* data dependency. This sequence of HDMs could be combined into one HDM rather than being computed in separate ones. During the fusion process, functions associated within the fusible HDMs, such as Map, Find, filter, local reduce/group will be directly appended to the parent nodes until it reaches the root or encounter an *N-to-N* and *One-To-N* dependency. The rule of function fusion in HDM can be specified as:

Function Fusion Rule: Given two connected HDMs: $HDM_1[T, R]$ with function $f : T \rightarrow R$ followed by $HDM_2[R, U]$ with function $g : R \rightarrow U$, if the dependency between them are one-to-one then they can be combined as $HDM_c[T, U]$ with function $f(g) : T \rightarrow U$.

This rule can be applied recursively on a sequence of fusible operations to get the final combined HDM. For example, after performing the function fusion, the data flow of `WordCount` can be simplified as shown in Fig. 7(c). Fusible operations such as Map and FlatMap are both fused into top DDMs and inherit the data dependency as *One-To-N*. During execution, the combined function is directly executed on every input block of data within one task.

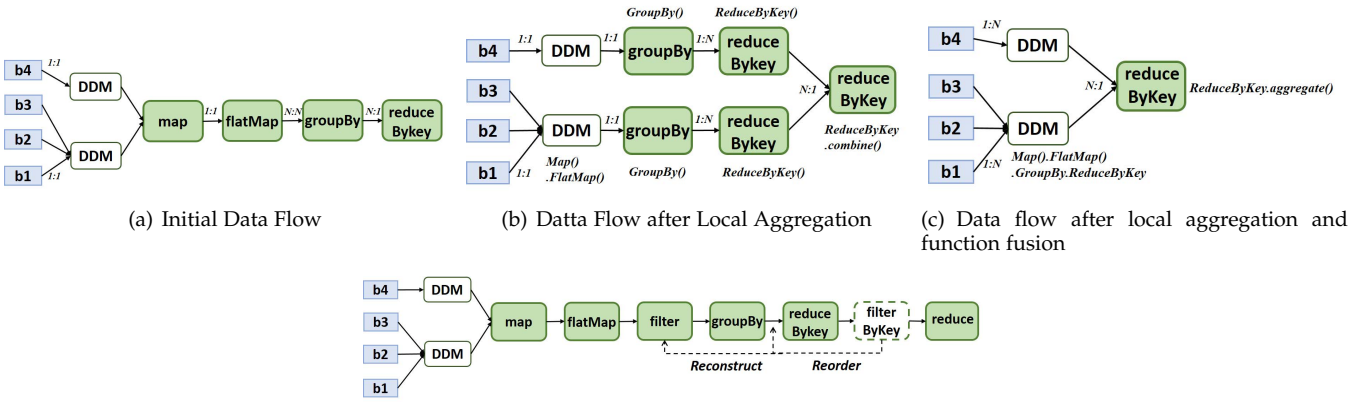
4.3 Re-ordering/Re-construction Operations

TABLE 4. Rewriting Patterns in HDM

Pattern	Re-written
map($m : T \rightarrow R$) .filter($f : R \rightarrow Bool$)	filter($f(m) : T \rightarrow Bool$) .map($m$)
union().filter($f : T \rightarrow Bool$)	filter($f : T \rightarrow Bool$).union()
intersection().filter($f : T \rightarrow Bool$)	filter($f : T \rightarrow Bool$) .intersection()
distinct().filter($f : T \rightarrow Bool$)	filter($f : T \rightarrow Bool$) .distinct()
sort().filter($f : T \rightarrow Bool$)	filter($f : T \rightarrow Bool$).sort()
groupBy($g : T \rightarrow K$) .filterByKey($f : K \rightarrow Bool$)	filter($f(g) : T \rightarrow Bool$) .groupBy($g$)
reduceByKey() .filterByKey($f : K \rightarrow Bool$)	filterByKey($f : K \rightarrow Bool$) .reduceByKey()

Apart from aggregation operations, there is another set of operations (like *Filter* or *FindBy*) that can reduce the total communication cost by extracting only a subset of data from previous input. These operations are considered as pruning-operations during execution. The basic principle is that

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2017.2690906, IEEE Transactions on Big Data



(a) Initial Data Flow (b) Datta Flow after Local Aggregation (c) Data flow after local aggregation and function fusion

(d) Data flow Reconstruction

Figure 7. Data Flow Optimizations on WordCount

the optimizer attempts to lift these pruning-operations to reduce the data size in advance while sinking the operations which involves global aggregations (such as global *Reducer* and *Grouper*) to delay intensive communication.

During the optimizing phase, operation re-ordering and re-construction are achieved through re-writing of operation patterns. The optimizer keeps checking the operation flow and detect possible patterns and re-writes them in optimized formats. Basic patterns and re-written formats in optimizers are listed in TABLE IV. The operation re-writing process can be recursively performed for multiple iterations to obtain an optimal operation flow. As an example, consider a extended `WordCount` program, in which the developer wants to find out the number of words that start with the letter '*a*'. The extended program has two additional functions: *FindByKey* and *ReduceByValue*, as shown in Fig. 7(d). During optimization, as indicated in Fig. 7(d), the *FindByKey* function is categorized as one of the pruning-functions that can reduce the total data amount. Thus, it is lifted before the aggregation *ReduceByKey*. When it meets the shuffle operation *GroupBy*, applying the same rule, it continues to be lifted until it reaches the parallel operation *FlatMap* and *Map*.

4.4 HDM Cache

For many complicated applications (such as iterative algorithms), input data might be repetitively used in the iterations. Therefore, it is necessary to cache this part of data to avoid redundant computation. In HDM, data caching can be triggered through two ways:

- Developers can explicitly specify which HDMs need to be cached for repetitive computation.
- Cache operation can be automatically added by doing reference counting in the logical planning phase, in which HDMs directly referenced by multiple subsequent operations will be labeled. The output of these HDMs will be cached for subsequent execution;

For example, as shown in Fig. 8, the sort operation contains four functions: map, sample, partition and sort. The sample function and partition function share the same input that is calculated from the map's output. In the logical planning phase, the planner can detect that the output of map has multiple references (with value of '2') thus the planner can add a cache tag on it. Then, during execution phase, all the output with this tag will be cached in the memory of workers to avoid redundant computations.

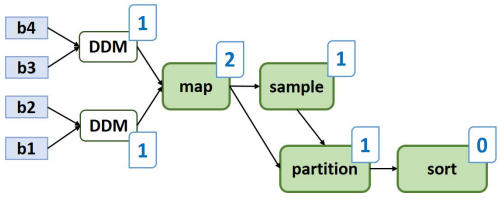


Figure 8. Cache Detecting by Reference Counting.

Once a HDM is labeled and cached by a previous job, in the next planning phase, the cache optimizer will check and replace all the references of this HDM in the data flow with the cached DDM addresses. During execution, there are two types of caching policies: eager caching and lazy caching. The former actively computes the caching data before it is required by subsequent operations; the latter will only starts computing the caching data until the first reading operations is invoked. By default, lazy caching is used during HDM computation.

4.5 Comparison with Optimizations in Other Frameworks

The optimization mechanisms introduced in this sections are not newly invented by us it is derived from some optimizations in other frameworks in combination with functional wrapping.

- In MapReduce, users can manually define Combinators for each Mappers to apply map-side merge in a MapReduce job In Spark, users can use Aggregator API rather than normal groupBy and reduce operations to define the aggregation operation before shuffling data across the cluster. Those are all derivations of the local aggregation optimizations.
- Tez combines the chained map-only jobs into one Mapper to reduce the complexity and consumption for executing multiple ones. Spark scheduler schedule a sequence of arrow dependency operations into one task to reduce the redundant cost for scheduling and generation of intermediate data. These optimizations achieve

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2017.2690906, IEEE Transactions on Big Data

similar effects as our function fusion optimization yet in a task-oriented perspective.

- Support of operation re-ordering and re-construction are not natively provided in current frameworks such as MapReduce and Spark. In practice, it is required for the job developers to have certain expertise for optimizing the sequence of operations in their programs.
- Both Spark and Flink provide operations for developers to explicitly cache datasets into memory to improve the performance of iterative jobs. However, for many complicated programs it is hard to manually decide when and where to cache the datasets if it is repeatedly used in subsequent operations in a implicit manner.

In contrast to manually applying most of those optimizations in existing frameworks, HDM utilizes the information and understanding of functional semantics to be able to automatically perform the optimization mechanisms in the data flow of the HDM jobs. More importantly, the automation of optimizations also make them easier and feasible to be applied to complicated jobs and data pipelines.

5 System Implementation

The kernel of the HDM runtime system is designed to support the execution, coordination and management of HDM programs. For the current version, only memory-based execution is supported in order to achieve better performance.

5.1 Architecture Overview

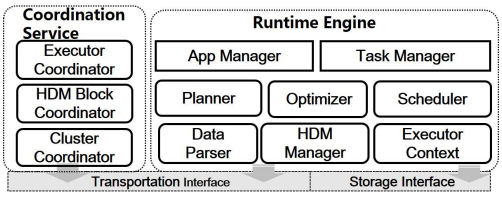


Figure 9. System Architecture of HDM Runtime System.

Fig. 9 illustrates the system architecture of HDM runtime environment which is composed of three major components:

- *Runtime Engine*: is responsible for the management of HDM jobs such as explaining, optimization, scheduling and execution. Within the runtime engine, App Manager manages the information of all deployed jobs. It maintains the job description, logical plans and data types of HDM jobs to support composition and monitoring of applications; Task Manager maintains the activated tasks for runtime scheduling in Schedulers; Planers and Optimizers interpret and optimize the execution plan of HDMs in the explanation phases; HDM Manager maintains the HDM information and states in each node of the cluster and they are coordinated together as an in-memory cache of HDM blocks; Executor 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 coordination: cluster coordination, HDM block coordination and executor coordination. They are responsible for coordination and management of node resources, distributed HDM blocks and distributed executions within the cluster context, respectively.
- *IO interface*: is a wrapped interface layer for data transfer, communication and persistence. IO interfaces are categorized as transportation interfaces and storage interfaces in implementation. The former is responsible for communications and data transportation between distributed nodes while the latter is mainly responsible for reading and writing data on storage systems.

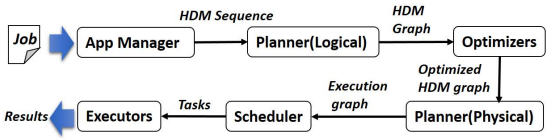


Figure 10. Process of executing HDM jobs.

In the following parts of this section, additional details about the major components are presented.

5.2 Runtime Engine

The main responsibility of the components in the runtime engine is the coordination and cooperation of tasks so that the jobs specified as HDMs can be completed successfully. Fig. 10 shows the main process of executing HDM jobs in the runtime system. As shown, the main phases for executing HDM jobs include logical planning, optimization, physical planning, scheduling and execution. Before execution, HDMs need to be explained as executable tasks for executors. The explanation process is mainly divided into two sub-steps: logical planning and physical planning.

5.2.1 Logical Planning

Algorithm 1: LogicalPlan

```
Data: a HDM h for computation
Result: a list of HDM List_h sorted by dependency
begin
    if children of h is not empty then
        for each c in children of h do
            List_h += LogicalPlan(c);
        end
        List_h += h;
    else
        return h;
    end
    return List_h;
end
```

In the logical planning step, a HDM program will be represented as a data flow in which every node is a HDM object that keeps the information about data dependencies, transformation functions and input output formats. The logical planning algorithm is presented in Algorithm 1. Basically, the planner traverses the HDM tree from the root node in a depth-first manner and extracts all the nodes into the resulting HDM list which contains all the nodes for a logical data flow. After the construction of the data flow, all the necessary HDMs will be declared and registered into the HDM Block Manager. In next step, optimizations will be performed on the logical data flow based on the rules

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2017.2690906, IEEE Transactions on Big Data

discussed in Section 4. So far, the logical data flow is still an intermediate format for execution. In order to make the job fully understandable and executable for executors, further explanation is required in the physical planning phase.

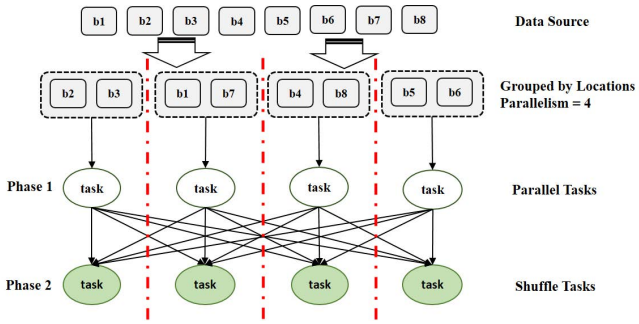


Figure 11. Physical execution graph of HDM (parallelism = 4).

5.2.2 Physical Planning

Algorithm 2: PhysicalPlan

```
Data: the root HDM h and the parallelism p expected for
      execution
Result: a list of HDM List_h sorted by dependency
begin
    if children of h is empty then
        l_s ← block locations of h;
        g_s ← clustering l_s as p groups according to
        distance;
        for each g in g_s do
            d ← create a new DFM with input g and
            function of h;
            List_h += d;
        end
    else
        for each c in children of h do
            List_h += PhysicalPlan(c);
        end
        d_s ← split h as p branches of DFM;
        list_h += d_s;
    end
    return List_h;
end
```

In the physical planning phase, given the logical data flow, the planner explains it further as a DAG task graph according to the parallelism on which the job is expected to be executed. The pseudo code of physical planning is presented in Algorithm 2. Essentially, physical planning is a low-level explanation which splits the global HDM operations into parallel ones according to the parallelism and data locations. As presented in Fig. 11, the physical planning phase contains three main steps:

- First, the data source is grouped/clustered based on location distance and data size.
- Second, the following operations are divided as multiple phases based on the boundary of shuffle dependencies.
- Last, parallel operations in each phase are split into multiple branches according to the expected execution parallelism (p=4). Then every branch in each phase of the graph is assigned as a task for follow-up scheduling and execution.

6 Dependency and History Management of HDM

During continuous development and deployment of big data analytic applications, it is normally tedious and complicated to maintain and manage applications that are constantly evolving. In HDM, by drawing on comprehensive information maintained by HDM models, the runtime engine is able to provided more sophisticated dependency and history management for submitted jobs.

6.1 History Traces Management

In HDM, once a application is submitted to the server, it will be assigned with an application ID(if it is not specified) and version number. Then, any future submissions and updates for the same application (identified the application ID) will obtain a new and unique auto-increased version number. In addition, execution dependencies (e.g. binary jars of the libs) are decoupled from execution programs for HDM. Before executing of a HDM application, all the dependent libraries that are associated with the specific application ID and version number that are required to be submitted to the server. Subsequently, the application can be execute at any time at any location within the cluster without re-submitting the dependencies. This facilitates the dependency management for applications and also makes the application re-producible and portable across developers and locations. Moreover, during execution of HDM jobs, all the meta data of the applications, such as the logical plan, optimized plan, physical plan and timestamp of tasks will be automatically recorded by the server. This information of jobs are very helpful for users to profile, debug and analyze their deployed applications and jobs in their life cycles. In the HDM server, it provides two basic data structures to maintain the meta data of HDM:

- Dependency Trace, which records the dependent libs required for execution of each version of the submitted applications.
- Execution Trace, which records information related to the runtime execution process of each task of HDM, including HDM ID/TaskID, createTime, scheduledTime, completionTime, input/output DDMs and input/output types.

All of these meta data that has been collected for HDM jobs are maintained as a tree-based history in the HDM server. Based on the two traces information stored on the server, users can apply queries to explore and analyze the historical information in which they are interested. Basically, users can query history information by sending two types of messages:

- *AppHistoryMsg*, which specifies the application Name and a version duration. The response contains the matched history information from the dependency trace trees.
- *ExecutionHistoryMsg*, which specifies the execution instance ID. The response contains execution DAG with all the task information that are related to the execution instances.

A concrete example about the historical trees maintained in HDM manager is shown in Fig. 12. Basically, after an application (e.g. `WordCount`) is submitted to HDM server, the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2017.2690906, IEEE Transactions on Big Data

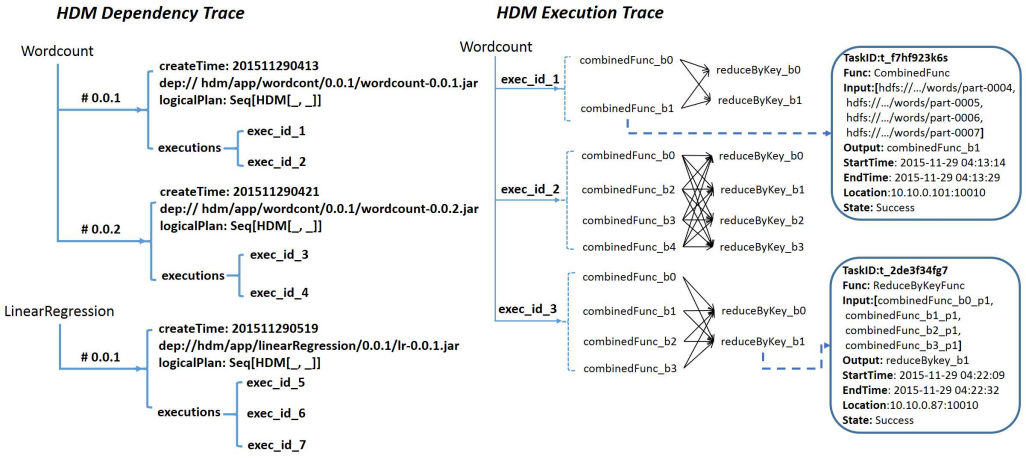


Figure 12. Dependency and Execution Traces of HDM.

dependency information such as (application ID, version, dependent libs and the original logical plan) are recorded in the dependency tree. In addition, every new version of the same application creates a new branch under the application root and is differentiated by the version number.

For each version of the application, the job can be executed for multiple times. Correspondingly, each execution of the job will generate a execution instance in the execution trace tree. In each execution trace, the physical execution DAG of the job is recorded. Furthermore, for each node within the DAG, the runtime information (including task ID, execution location, input/output paths and execution states) are also automatically maintained. Additionally, the references of execution instances are also maintained in the dependency trees accordingly (under each branch version of the applications) to facilitate future queries.

6.2 Reproduce of HDM Applications

Drawing on the logical plan history recorded in the manager, developers or data scientists are able to reproduce the historical HDM jobs and applications when requested (assume the input data is still available). This is a very significant feature for learning-based applications and algorithms because there might be multiple back-and-forth steps during exploring a data set and finding the most suitable algorithm and parameters. In order to re-execute a historical HDM application, an user needs to specify the application name and its version as listed in Fig. 13:

```
wordcount = HDM[(String,Int)]("hdm://wordcount", 0.0.1)
wordcount.compute(context = "10.10.0.100:8999")
onComplete {
      case Success(resp) ⇒ resp.foreach(println)
      case Failure(exception) ⇒ println(exception)}
```

Figure 13. Reproducing an existing word-count application

6.3 Composition of HDM applications

In addition to reproducing a historical application, developers can also apply composition with existing applications in HDM. Consider the `WordCount` program (Fig. 4) as an example. Assume the program is already deployed on the HDM server, a subsequent developer wants to write an extended program which counts the sum of all words that start with the letter *'a'*. In HDM, as we already have a `WordCount` program deployed, the subsequent developer just needs a few lines code to write a follow-up program to filter the `WordCount` results which start with the letter *'a'* (as shown in Fig. 14). By specifying the data format, URL and expected version of an existing HDM job/application, developers can directly use it and re-generate extended jobs and applications.

```
wordcount = HDM[(String,Int)]("hdm://wordcount", 0.0.1)
newCount = wordcount.filterByKey(k ⇒ k.startsWith("a"))
```

Figure 14. Applying a new operation to an existing program

Compared to HDM, the common approach in MapReduce or Spark is to re-write a brand new job and re-deploy it to the cluster. However, this is not the most elegant way to achieve composability. In HDM, as the logical plans of all the deployed applications are automatically maintained in the server, the application manager is able to apply composition based on the functional DAGs of applications rather than requiring developers to rewrite the entire program manually.

An additional example is that sometimes developers want to reuse the same program to process data sets from different sources with different formats. In HDM, developers can re-use an existing job with a new HDM as input. Consider the `WordCount` example, we want to process a different data source in which words are separated by *semicolons* rather than *commas*. As the previous HDM program (in Fig. 4) is a HDM with the input type of *String*, a substitute input can be constructed from the new data source and passed through to replace the old input HDM. To achieve this, developers only need to write a few lines of code as shown in Fig. 15.

The two examples in this section also refer to the two basic composition patterns of HDM, Compose and AndThen respectively (described in section 3.2). Composability of HDMs significantly improves development efficiency and is also very meaningful for integrations and continuous

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2017.2690906, IEEE Transactions on Big Data


```
newText = HDM.string("hdfs://10.10.0.1/newPath"))
          .flatMap(_.split(";"))
newCount = HDM[(String,Int)]("hdm://wordcount", 0.0.1)
          .compose(newText)
```

Figure 15. Replacing the input of an existing program

development because one team of developers can easily share their mature and fully tested data-driven applications. During the execution process, composed HDMs are new job instances and will be executed separately with no interference with the original ones. However, as HDM is strongly-typed, one assumption for the composition of HDM applications is that subsequent developers should know about the data types and URL in order to reuse an existing HDM job.

6.4 Fault Tolerance in HDM

Draw on the dependency graph and the reproducibility of HDM jobs, current HDM execution engine provides fault tolerance for data processing by using pushing lineage, in which the lost or failed data partitions would be re-compute from its parents or ancestors in the data dependency graph. Pushing lineage is a well known technique that is also used in modern data-intensive frameworks such as Spark, Tachyon, Nectar and BAD-FS. In the future, we are planing to add more fault tolerance mechanisms such as replication and snapshotting to support the requirements for different types of applications.

7 Evaluation

7.1 Case Study

In the first half of the evaluation, a case study is presented to show how users can develop a pipelined machine learning application by integrating multiple HDM jobs. We use the Image Classification Pipeline as the case study, in which the training pipeline consists of three main parts: the image data parser, feature extractor and classification trainer.

```
/* define model */
model = new AtomicObject[Vector]
/* specify data */
data = HDM[_, Byte]("hdfs://10.10.0.1:9001/images/*"))
       .map(arr ⇒ Vector(arr))
/* specify existing components */
extractor= HDM[Vector,Vector]("hdm://feature-extractor", 1.1)
learner= HDM[Vector,Vector]("hdm://linear-classifier", 1.0.0)
/* connect components as a pipelined job
*/
imageApp = extractor.compose(data).andThen(learner)
/* run job and update the model */
imageApp.traverse(context="10.10.0.1:8999") onComplete {
       case Success(resp) ⇒ model.update(resp.next)
       case Failure(exception) ⇒ println(e)
}
```

Figure 16. Creating an image classification pipeline in HDM

In the image-classification pipeline, components like feature extractor and classification trainer are commonly-used algorithms for many machine learning applications. This means that they could be developed by other developers and published as shared applications in HDM. Then, subsequent developers can find those exiting HDM descriptions of which they can directly re-use and integrate them to create a simple image classification application by writing just a few lines of code as shown in Fig. 16.

In principle, the kernel engine of Hadoop and Spark does not support direct integration and composition of deployed jobs for subsequent development. Therefore, programmers may need to manually combine programs written by different developers or re-write every components by themselves. High level Frameworks such as Flume[1], Pig [8] and Oozie [6] support writing data pipelines in a re-defined programming manner and automatically generates MapReduce jobs. However, it sacrifices some flexibility for integration and interaction with the general programming context of developers. For HDM programs, as they are essentially a programming library of Scala, users can directly embed HDM codes within other Scala or Java programs. Basically, the HDM server is acting as both an execution engine and application repository which enables developers to easily check and integrate with published HDM applications.

7.2 Performance Evaluation

In this section, we show the results of comparing the performance of HDM with Apache Spark (version 1.4.1).

7.2.1 Experimental setup

Our experiments have been applied on Amazon EC2[2] with one M3.large instance as master and 20 M3.2xlarge (with 8 vCPUs - Intel Xeon E5-2670 v2 Processors, 30GB memory, 1Gib network) instances as workers. Every Spark and HDM worker is running in the memory-only model on the JVM with 24 GB memory - 480 GB aggregated JVM memory for the entire cluster so that all the data can be fed into the main memory during execution of our test cases. In addition, data compression options in Spark are all disabled to avoid performance interference.

For testing data sets, the user-visits from AmpLab [3] is used for the primitives and SQL benchmarks. The original user-visits data set is 119.34 GB, we replicate it to 238.68GB for larger scale tests. The Daily Global Weather Measurements (DGWM) data [4] from AWS public repository is used for the machine learning benchmark. The DGWM data is also replicated to 61.30GB and 122.60GB to match the large scale testing scenario in our experiments. Before running experiments, all tested data sets are downloaded and persisted in the HDFS (Hadoop 2.4.0) which is hosted on the same cluster for executing the tested jobs.

7.2.2 Experimental Benchmarks

During experiments, we try to comprehensively compare the performance of HDM and Spark for different types of applications. The test cases are categorized into four groups: basic primitives, pipelined operations, SQL queries and Machine Learning algorithms as listed below (TC-i denotes the i-th test case).

1. http://incubator.apache.org/flume
2. https://aws.amazon.com/ec2/
3. https://amplab.cs.berkeley.edu/benchmark/
4. http://aws.amazon.com/datasets/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2017.2690906, IEEE Transactions on Big Data

Basic Primitives: Simple parallel, shuffle and aggregation operations are tested to show the basic performance of the primitives for both Spark and HDM:

- TC-1, *Simple parallel operation*: A Map operation which transforms the text input into string tuples of page URL and value of page ranking;
- TC-2, *Simple shuffle operation*: A GroupBy operation which groups the page ranking according to the prefix of page URL;
- TC-3, *Shuffle with map-side aggregation*: A ReduceByKey operation which groups the page ranking according to the prefix of page and sums the total value of every group.
- TC-4, *Sort*: The implementation of Sort operation contains two phases: in the first phase input data are sampled to get the distribution boundary for range partitioning in next phase; in the second phase, input data are partitioned using the range partitioner then each partitioned block is sorted in parallel.

Pipelined Operations: The performance of complicated applications can be derived from the basic performance of meta-pipelined ones. In the second part of our experiments, the typical cases of meta-pipelined operations are tested, respectively, by each of the following test cases:

- TC-5, *Parallel sequence*: Five sequentially connected Map operations each of which transforms the key of the input into a new format.
- TC-6, *Shuffle operation followed by aggregation*: A *groupBy* operation which groups the page ranking according to the prefix of page; then followed with a Reduce operation to sum the total values of every group.
- TC-7, *Shuffle operation followed by filter*: The same *groupBy* operation as TC-6, then followed with a filter operation to find out the ranking that starts with a certain prefix.
- TC-8, *Shuffle operation followed by transformation*: A *groupBy* operation which groups the page ranking according to the prefix, then appends with a map operation to transform the grouped results into new formats.

SQL queries: SQL-like operations is crucial for data management systems. Therefore, most data processing framework provide SQL interfaces to facilitate data scientists in their task of applying ETL (Extraction, Transformation, Loading) operations. To evaluate the performance of SQL operations, we test HDM's POJO based ETL operations against SparkSQL.

- TC-9, *Select*: Selects a subset of columns for the tabular data, a *Select* operation is actually achieved by one parallel map operation;
- TC-10, *Where*: Scan the data to find out records that meet the condition expression;
- TC-11, *OrderBy*: Order the input data set by given column.
- TC-12, *Aggregation with groupBy*: groupBy the records and aggregate the value of a numeric column in each group;

Iterative Algorithms: Many data analytic applications such as machine learning algorithms are iterative jobs. To evaluate the performance of iterative jobs, we test two commonly used machine learning algorithms (Linear Regression and K-Means) for Spark and HDM:

- TC-13, *Linear Regression*: Apply linear regression based on stochastic gradient descent;
- TC-14, *K-Means*: Apply K-means clustering on the input data, in the experiment, we choose K=128;

For both test cases of Linear Regression and K-Means, we use the example implementation in the official Spark-example component and provide equivalent implementation using HDM primitives.

7.2.3 Experiment Results

We compared the Job Completion Time (JCT) of the above tested cases for HDM and Spark. The results of each tested group are discussed as follows.

- Comparison of Basic Primitives

Simple parallel operation: As shown in TC-1 in the Fig. 17, for the parallel operations like Map, HDM has similar performance as Spark. However, HDM is slightly faster, as it provides a global view in the planning phases so that workers are assigned with parallel branches in the execution graph rather than as a single step. In such a case, less communication is required for completing the same job. Besides, the Spark scheduler uses delay scheduling to achieve better data localization by default. For HDM, data localization and fairness are pre-considered in the physical planning phase, so no delay is required during scheduling tasks, which also decreases the latency of assigning tasks to some extent.

Simple shuffle operation: For the *GroupBy* operation in TC-2, HDM shows 15% shorter JCT compared to Spark. Basically, HDM benefits from the parallel aggregation and de-coupled implementation of IO and computation during the shuffle stage.

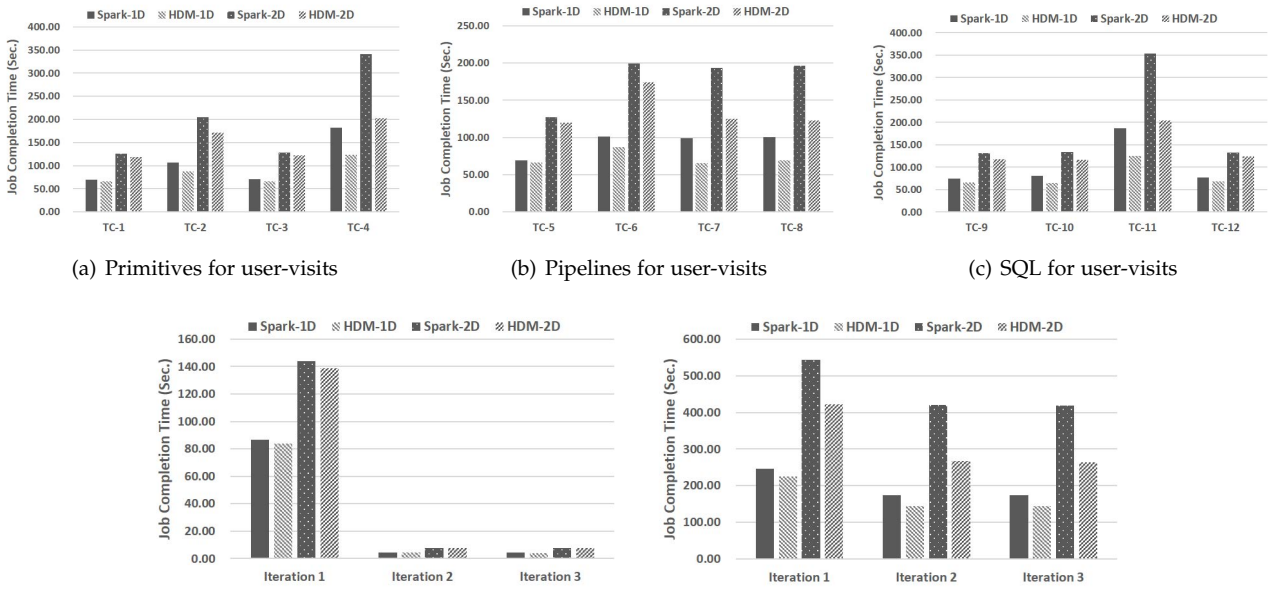
Shuffle operation with map side aggregation: Spark provides several optimized primitives to improve the performance for some shuffle-based aggregations by using map-side aggregators e.g. ReduceByKey. In HDM, this type of optimization is achieved by automatically applying local aggregation when necessary. Eventually, for optimized shuffle operations, HDM still achieves slightly better performance (TC-3) than Spark, which is mainly due to the better efficiency in the scheduling phases.

Sorting: For general sort operation, both Spark and HDM contain the same steps (sample the data, range partitioning then parallel sorting). However, HDM shows more than 30% improvement (in TC-4) by detecting and caching the input data after the sample step. As the whole input data is already loaded into memory during the sample step, HDM caches the data for the subsequent partitioning step. For Spark, the partitioning step needs to reload the input data from HDFS again.

- Comparison of Pipelined Operations

Multiple parallel operations: For multiple parallel operations (TC-5), Spark wraps sequentially connected operations into one task, within which pipelined operations are executed one by one iteratively. For HDM, the optimization is applied by parallel function fusion, in which multiple operations are merged into one high-order function and applied only once for every record of the input. As a result,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2017.2690906, IEEE Transactions on Big Data



(a) Primitives for user-visits (b) Pipelines for user-visits (c) SQL for user-visits

(d) Linear Regression for Weather Data set (e) KMeans for Weather Data set

Figure 17. Comparison of Job Completion Time for HDM and Spark
Note:`Spark-1D`, `HDM-1D` and `Spark-2D`, `HDM-2D` denote the results for initial dataset and double-sized dataset, respectively.

the JCT of both Spark and HDM does not increase too much compared to single parallel primitives (TC-1). However, HDM shows slightly better performance than Spark due to less scheduling latency.

Shuffle operation followed by data transformation: HDM and Spark both show relatively longer JCT for shuffle operations followed with general transformations (TC-6). Without sufficient semantics about the user-defined transformation function, no optimization was applied by both HDM and Spark. Thus, the results for this test case is quite similar to the basic shuffle primitives. HDM generates shorter JCT due to more efficient parallel aggregations in shuffling.

Shuffle operation followed by aggregation: For operations composed by shuffle and aggregation (TC-7), HDM shows much better performance (35-40%) than Spark. This is because there is no data flow analysis in the core engine of Spark whereas HDM can recognize aggregations behind shuffle operations and automatically apply re-ordering and local aggregation to reduce the size of data required for subsequent shuffling.

Shuffle operation followed by pruning operation: For operations sequences which contains pruning operations (TC-8), HDM also achieves considerable improvement by being able to perform re-ordering and re-writing to push pruning operations forward as much as possible. Therefore, the data size of following data flow is significantly reduced.

- Comparison of SQL Queries

For basic SQL queries, SparkSQL uses Catalyst to optimize their execution plan while HDM relies on the built-in HDM optimizations. As a result, the performance is quite close for `Select` (TC-9), `Where` (TC-10) and `Aggregation` (TC-12) as shown in Fig. 17(c). HDM indicates around 10% shorter JCT for those test cases. However, for the `orderBy` query, SparkSQL does not apply any optimizations for sorting whereas HDM adds caching after loading the input data. Therefore, HDM shows around 30% improvement in TC-13.

- Comparison of Iterative Jobs

Linear Regression: For the performance of linear regression, the example code implementation uses SGD (Stochastic Gradient Decent) to train the data. In SGD, every partition of data only needs to compute and send the local coefficiency vector for the final aggregation step. Apart from caching the input data, there is very little space for data flow optimizations. For this test case, both Spark and HDM caches the input data for the iterative learning process. As a result (in Fig. 17(d)), the performance of Spark and HDM are very close - after caching the data in the first iteration, it becomes much faster (20x) in the subsequent iterations.

KMeans: For K-Means clustering, it contains more intensive computation and data transferring steps than linear regression. For each iteration, the job needs to compute to the square distance between each candidate and every point in the candidate cluster is then updated with the new candidates. In the implementation, a *map-reduceByKey-map* pipeline is involved in each iteration. Similarly, both Spark and HDM cache the input data for iterative learning. The results (in Fig. 17(e)) show that subsequent iterations of both Spark and HDM gain about 30% JCT reduction after caching the data in the first iteration. However, HDM shows around 10% shorter JCT for the first iteration and 20% shorter JCT for subsequent iterations compared to Spark. Basically, HDM benefits from better performance in pipelined executions as discussed in previous test cases.

7.2.4 Discussion and Summary

In our experiments, HDM generally shows better performance than Spark. However, we understand that there are a bunch of parameters that can be tuned in Spark to obtain better performance. In addition, the results might be bounded by the coverage of test cases that we choose and some optimizations are based on the assumption that the input data can be fully loaded into the memory of the cluster. Despite of these limitations, the experiments in

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2017.2690906, IEEE Transactions on Big Data

TABLE 5. COMPARISON OF MAJOR BIG DATA FRAMEWORKS

	MR	Spark	Flink	HDM
Data Model	key-value	RDD	flat record	HDM
Programming Model	Map-Reduce	functional	functional	functional
Job Planing	task-based	DAG-based	DAG-based	DAG-based
Data Flow Optimization	N/A	manual	auto (inherit from DBMS)	auto (DAG & functional)
Composition and pipeline	high level frameworks	high level frameworks	high level frameworks	native support

this section are not supposed to show which framework is essentially better but to illustrate that a data processing engine can significantly benefit from optimizations based on the functional DAG flow. And these optimizations can be well applied automatically on a well defined data and execution model, such as HDM.

8 RELATED WORK

8.1 Big data processing frameworks

Several frameworks have been developed for providing distributed big data processing platforms [13]. MapReduce [4] is a commonly used big data processing paradigm which pioneered this domain. It uses key-value pairs as the basic data format during processing. Map and Reduce are two primitives which are inherited from functional programming. In terms of performance, Hadoop/Map-Reduce jobs are not guaranteed to be fast. All the intermediate data during execution are written into a distributed storage to enable crash recovery. This is a trade-off which sacrifices the efficiency of using memory and local storage. The Map-Reduce framework is usually not effective for fast and smaller jobs where the data can fit into memory [11].

Spark [16] utilizes memory as the major data storage during execution. Therefore, it can provide much better performance when compared with jobs running on MapReduce. The fundamental programming abstraction of Spark is called *Resilient Distributed Datasets* (RDD) [15] which represents a logical collection of data partitioned across machines. Besides, applications on Spark are decomposed as Stage-based DAGs that are separated by shuffle-dependencies. In job explanation, Spark also combines parallel operations into one task, in which sense, it also achieves the similar optimization of function fusion as that in HDM. However, further data sequence optimizations such as operation reordering and rewriting are not provided in the Spark processing engine.

Apache Flink[5] has originated from the Stratosphere project [1] which is a software stack for parallel data analysis. Flink shares many similar functionalities with Spark and it provides optimizations that have been inspired from relational databases but adapted to schemaless user defined functions (UDF). Compared with Spark, Flink has a pipelined execution model which is considered better suited to iterative and incremental processing.

The main difference between HDM and other functional like frameworks (Spark and Flink) is the programming abstraction. RDD in Spark is a data-oriented abstraction - each RDD represents a distributed dataset with a certain type. Flink shares similar concept as Spark, it uses DataSet object which is also an abstraction of distributed datasets. In comparison, HDM is a function-oriented abstraction - each HDM is a wrapper of a function which takes a set of input dataset and compute the output dataset. Both Spark and Flink has function objects during execution but those functions are supplementary to their data abstractions. In HDM functions are the top one citizen while datasets are supplementary (can be replaced just as change the input of a function) to functions. Due to the function-oriented nature, HDM is able to natively utilize functional optimization and composition mechanisms to improve the execution performance and provide some extent of reusability. TABLE 5 compares the main features between HDM and the major open-source big data frameworks.

8.2 Optimizations for Big Data Applications

FlumeJava [3] is a java library that was recently introduced by Google. It is built on MapReduce and provides a higher level wrapping and a bunch of optimizations for better execution plans. The performance benefits of those optimized plans are quite close to manually optimized MapReduce jobs but FlumeJava has released programmers from the redundant and often tedious optimization process. Apache Pig [8] provides some extent of reusability by supporting registering and loading User Defined Functions in pig scripts. HDM provides the job reusability by supporting composition of deployed jobs and also maintaining the evolving history and dependency for each of them. In addition, HDM server is acting as a shared repository for all the deployed HDM programs while developers need to separately maintain and synchronize the Pig UDF functions for their developing and execution environment.

Tez [9] is a graph-based optimizer which can significantly optimize MapReduce jobs written in Pig [8] and Hive [5]. Basically, Tez simplifies the MapReduce pipelines by combining multiple redundant Mapper and reducers. It also directly connects the outputs of previous jobs to following ones, which reduces the cost of writing intermediate data into HDFS and thus can improve the performance of executed pipelined jobs. *Apache MRQL*[6] is a framework that has been introduced as a query processing and optimization framework for distributed and large-scale data analysis, built on top of Apache Hadoop, Spark, Hama, and Flink. In particular, it provides an SQL-like query language that can devaluated in four *independent* modes: MapReduce mode using Apache Hadoop, Spark mode using Apache Spark, BSP mode using Apache Hama, and Flink mode using Apache Flink. *Apache Crunch* [7] is Java library that provides implementing pipelines that are composed of many user-defined functions and can be executed on top of Hadoop and Spark engines. Apache Crunch is based on Google's FlumeJava library [3] and is efficient for implementing common tasks like joining data, performing aggregations, and sorting records. *Cascading* [8] is another software abstraction layer for the Hadoop framework which is used to create

5. https://flink.apache.org/

6. https://mrql.incubator.apache.org/
7. https://crunch.apache.org/
8. http://www.cascading.org/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2017.2690906, IEEE Transactions on Big Data

and execute data processing workflows on a Hadoop cluster using any JVM-based language (Java, JRuby, etc.), hiding the underlying complexity of the Hadoop framework.

9 Conclusion and Future Work

In this paper, we have presented HDM as a functional and strongly-typed meta-data abstraction, along with a runtime system implementation to support the execution, optimization and management of HDM applications. Based on the functional nature, applications written in HDM are natively composable and can be integrated with existing applications. Meanwhile, the data flows of HDM jobs are automatically optimized before they are executed in the runtime system. In addition, programming in HDM releases developers from the tedious task of integration and manual optimization of data-driven programs so that they can focus on the program logic and data analysis algorithms. Finally, the performance evaluation shows the competitive performance of HDM in comparison with Spark especially for pipelined operations that contains aggregations and filters.

We would like to note hat HDM is still in its initial stage of development, of which some limitations are left to be solved in our future work: 1) disk-based processing needs to be supported in case the overall cluster memory is insufficient for very large jobs; 2) fault tolerance needs to be considered as a crucial requirement for practical usage; 3) one long-term challenge we are planning to solve is about the optimizations for processing heterogeneously distributed data sets, which normally cause heavy outliers and seriously slow down the overall job completion time and degrade the global resource utilization.

References

[1] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. The Stratosphere platform for big data analytics. *VLDB J.*, 23(6), 2014.

[2] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: Relational Data Processing in Spark. In *SIGMOD*, pages 1383–1394, 2015.

[3] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. FlumeJava: easy, efficient data-parallel pipelines. In *PLDI*, 2010.

[4] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1), 2008.

[5] Yin Huai, Ashutosh Chauhan, Alan Gates, Günther Hagleitner, Eric N. Hanson, Owen O'Malley, Jitendra Pandey, Yuan Yuan, Rubao Lee, and Xiaodong Zhang. Major technical advancements in Apache Hive. In *SIGMOD*, pages 1235–1246, 2014.

[6] Mohammad Islam, Angelo K. Huang, Mohamed Battisha, Michelle Chiang, Santhosh Srinivasan, Craig Peters, Andreas Neumann, and Alejandro Abdelnur. Oozie: towards a scalable workflow management system for hadoop. In *SIGMOD Workshops*, 2012.

[7] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD Conference*, 2010.

[8] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, 2008.

[9] Bikas Saha, Hitesh Shah, Siddharth Seth, Gopal Vijayaraghavan, Arun C. Murthy, and Carlo Curino. Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications. In *SIGMOD*, 2015.

[10] Sherif Sakr and Mohamed Medhat Gaber, editors. *Large Scale and Big Data - Processing and Management*. Auerbach Publications, 2014.

[11] Sherif Sakr, Anna Liu, and Ayman G. Fayoumi. The family of mapreduce and large-scale data processing systems. *ACM CSUR*, 46(1):11, 2013.

[12] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, and Michael Young. Machine learning: The high interest credit card of technical debt. In *SE4ML: Software Engineering for Machine Learning*, 2014.

[13] Chun Wei Tsai, Chin Feng Lai, Han Chieh Chao, and Athanasios V. Vasilakos. Big data analytics: a survey. *Journal of Big Data*, 2(21), 2015.

[14] Dongyao Wu, Sherif Sakr, Liming Zhu, and Qinghua Lu. Composable and Efficient Functional Big Data Processing Framework. In *IEEE Big Data*, 2015.

[15] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*, 2012.

[16] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *HotCloud*, 2010.

Dongyao Wu is a PhD candidate in University of New South Wales (UNSW) and National ICT Australia (NICTA). He received his B.Sc. degree in Tianjin University in 2007 and accomplished his M.Sc. degree in the Institute of Software, Chinese Academy of Sciences in 2012. Before started his PhD in 2014, he was a senior software engineer in Baidu (Beijin, China) Inc. His research interests include big data infrastructure, distributed systems and parallel computing.

Sherif Sakr is an Associate Professor in the department of Health Informatics at King Saud bin Abdulaziz University for Health Sciences, Saudi Arabia. He is also affiliated with University of New South Wales (Australia) and at National ICT Australia (NICTA). Previously, He had appointments with Macquarie University (Australia), Microsoft Research (USA), Alcatel Lucent Bell Labs and Cairo University (Egypt). He received his PhD degree in Computer and Information Science from Konstanz University, Germany in 2007. He received his B.Sc. and M.Sc. degrees in Computer Science from the Faculty of Computers and Information in Cairo University, Egypt, in 2000 and 2003 respectively. Dr. Sakr is an IEEE Senior Member.

Liming Zhu is the Research Director of Software and Computational Systems Research Program at Data61, CSIRO. He holds conjoint professor positions at University of New South Wales (UNSW) and University of Sydney. He formerly worked in several technology lead positions in software industry before obtaining a PHD degree in software engineering from UNSW. He is a committee member of the Standards Australia IT -015 (system and software engineering) group and IT-038 (Cloud Computing) group and contributes to the ISO/SC7/WG42 on architecture related standards. He has published more than 120 peer-reviewed conference and journal articles. His research interests include software architecture, dependable systems and data analytics infrastructure.

Qinghua Lu is a lecturer at Department of Software Engineering, China University of Petroleum, Qingdao, China. She received her PhD from University of New South Wales (UNSW) in 2013. Her research interests include software architecture, dependability of cloud computing, big data architecture, and service computing.