Composable and Efficient Functional Big Data Processing Framework

Dongyao Wu∗†, Sherif Sakr∗‡, Liming Zhu∗†, Qinghua Lu‡§
*Software Systems Research Group, NICTA, Sydney, Australia
‡School of Computer Science and Engineering, University of New South Wales, Sydney, Australia
§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 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 of between 10% to 60% of the Job-Completion-Time for different types of operation sequences when compared with the current state art, Apache Spark.

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

I. INTRODUCTION

Over the past years, several frameworks (e.g. MapReduce [3] and Spark [9]) 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 practice, many real-world scenarios require pipelining and integration of multiple big data jobs. For example, an image analysis application requires many pre-processing steps such as image parsing and feature extraction while the core machine learning algorithm is only one component within the whole analytic workflow. Current frameworks for building pipelined big data jobs mainly rely on high level frameworks (such as Flume[4], Pig Latin [5] and Oozie[6]) that are built on top of core process engines (Hadoop[7]) and use an external persistent service (HDFS [8]) for exchanging data. In principle, a main limitation in existing frameworks such as MapReduce and Spark is that jobs are roughly defined and packaged as executable jars and then executed as a black-box without exposing any of the functionalities. As a result of this, deployed jobs are not natively composable and reusable for subsequent development and integration. Additionally, losing functionality and dependency information for these black-boxed jobs also hampers the ability to perform possible optimizations on the data flow of job sequences to achieve better performance.

We believe that by improving the basic data and task models, the compositability and optimization problems could be addressed to great extent on the big data execution engine level. In this paper, we present the Hierarchically Distributed Data Matrix (HDM) along with the system implementation to support the writing and execution of composable and interactive big data applications. HDM is a light-weighted, 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. Exploiting the functional nature of HDM enables deployed applications of HDM to be natively composable 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. In particular, the main contributions of this paper can be summarized as follows:

- HDM, a lightweight, functional, strongly-typed data representation for developing and describing data-parallel applications.
- A runtime framework to support the execution and integration of HDM applications on distributed environments.
- A set of optimizations which is based on functional data dependency graph to improve the execution performance of HDM jobs.
- Experimental evaluation for the performance of HDM compared with the current state of art for big data processing systems, Apache Spark.

The remainder of this paper are organized as follows. Section II introduces the representation and basic features of HDM. Section III describes the programming model of HDM. Section IV presents the major optimizations applied to the HDM data flow. Section V presents the system architecture and implementations of the HDM runtime system. In section VI, we present a case study of writing pipelined applications in HDM and compare the performance of HDM with Apache Spark. In Section VII, we discuss the related work before we conclude the paper in Section VIII.

†School of Computer Science and Engineering, University of New South Wales, Sydney, Australia
‡School of Computer Science and Engineering, University of New South Wales, Sydney, Australia
§College of Computer and Communication Engineering, China University of Petroleum, Qingdao, China
TABLE I. ATTRIBUTES OF HDM

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The identifier of a HDM. It must be unique within each HDM context.</td>
</tr>
<tr>
<td>InType</td>
<td>The input data type of computing this HDM.</td>
</tr>
<tr>
<td>OutType</td>
<td>The output data type of computing this HDM.</td>
</tr>
<tr>
<td>Category</td>
<td>The node type of this HDM. It refers to either DFM or DDM.</td>
</tr>
<tr>
<td>children</td>
<td>The source HDMs of a HDM. It describes from where this HDM can be computed.</td>
</tr>
<tr>
<td>distribution</td>
<td>The distribution relation of children blocks, including horizontal and vertical.</td>
</tr>
<tr>
<td>dependency</td>
<td>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.</td>
</tr>
<tr>
<td>function</td>
<td>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.</td>
</tr>
<tr>
<td>blocks</td>
<td>The data blocks of this HDM. For DFM it can be an array of ID for children DDM. This field is only available after all children of this HDM are computed.</td>
</tr>
<tr>
<td>location</td>
<td>It refers to the URL address of this HDM on local or remote nodes. For DDM, the actual data are loaded according to the protocols in the URL such as hdfs://, file://, mysql:// and hdm://.</td>
</tr>
<tr>
<td>state</td>
<td>Current state of this HDM. A HDM can exist in different states such as Declared, Computed and Removed.</td>
</tr>
</tbody>
</table>

II. DATA REPRESENTATION OF HDM

As data representation is the core of a data processing system, we first introduce our data model called Hierarchically Distributed Data Matrix (HDM) which is a functional, strongly-typed representation for parallel data processing.

A. 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, location, distribution to support optimization and execution. The attributes of a HDM are listed in TABLE I. Based on these attributes, 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 all about the evaluation of the contained function on the input dataset (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. Operations and compositions of HDM are required to guarantee the compatibility of data types.
- **Portable**: A HDM is an independent object contains complete information for a computation task. Therefore, in principle, 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 applying optimization for data localizations to some extent during the planning phases.

B. Categorization of HDM

In principle, HDM is a tree-based structure which consists of two types of nodes:

- **Distributed Data Matrix (DDM)**: The leaf-nodes in a HDM hierarchy hold the actual data and are responsible for performing atomic operations on data blocks.
- **Distributed Functional Matrix (DFM)**: The non-leaf nodes hold both the operational and distribution of relations for children HDMs; during execution, it is also responsible for collecting and aggregating the results from children nodes when necessary.

From the functional perspective, a DDM is considered as a function which maps a path to an actual dataset. 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, DFM 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.

C. Data Dependencies of HDM

In principle, data dependencies between HDMs is an important aspect. In particular, by performing operations on HDM, data dependencies are implicitly added between pre- and post HDM nodes. Basically, there are four types of dependencies in HDM:

- **One-To-One** (1:1): The dependency between current HDM and input is parallel; the dependency between output and the subsequent HDM is also parallel.
- **One-To-N** (1:N): The dependency between current HDM and input is parallel; the dependency between output and the subsequent HDM involves shuffling and partitioning.
- **N-To-One** (N:1): The HDM depends on multiple partitions from the input HDMs; the dependency between output and the subsequent HDM is parallel.
- **N-To-N** (N:N): The current HDM depends on multiple partitions from input HDMs; the dependency between output and the subsequent HDM involves shuffling and partitioning.

In practice, data dependency information represent a crucial aspect during both execution and optimization in order to decide how a HDM is computed, and which optimizations can be applied on the data flow, if at all.

III. HDM PROGRAMMING

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.
Table II. Semantics of Basic Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null</td>
<td>Do nothing but return the input.</td>
</tr>
<tr>
<td>Map ( f : T \rightarrow R )</td>
<td>( F_p : \text{List}[T] \rightarrow \text{List}[R] )</td>
</tr>
<tr>
<td>GroupBy ( f : T \rightarrow K )</td>
<td>( F_d : \text{List}[T] \rightarrow \text{List}[K] )</td>
</tr>
<tr>
<td>Reduce ( f : (T, T) \rightarrow T )</td>
<td>( F_c : \text{List}[T] \rightarrow \text{List}[T] )</td>
</tr>
<tr>
<td>Filter ( f : T \rightarrow \text{Bool} )</td>
<td>( F_f : \text{List}[T] \rightarrow \text{List}[T] )</td>
</tr>
</tbody>
</table>

A. HDM Functions

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

\[
F_p : \text{List}[T] \rightarrow \text{List}[R] \quad (1)
\]

\[
F_a : (\text{List}[T], \text{List}[R]) \rightarrow \text{List}[R] \quad (2)
\]

\[
F_c : \text{List}[R] \rightarrow \text{List}[R] \quad (3)
\]

\( F_p \) 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. Normally, functions are required to be performed on multiple data partitions when the input is too large to fit into one task. The aggregation semantics are very useful under such 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-and-combine pattern.

During the explanation of HDM jobs, different semantics are automatically chosen by planers to hide users from functional level optimizations. To better explain the three types of semantics described above, an illustration of the semantics of some basic HDM functions are listed in Table II.

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 1.

B. HDM Composition

1) Composition patterns: In functional composition, one function \( f : X \rightarrow Y \) can be composed with another function \( g : Y \rightarrow Z \) to produce a second-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:

\[
\text{HDM}[T, R] \quad \text{compose} \quad \text{HDM}[I, T] \Rightarrow \text{HDM}[I, R] \quad (4)
\]

\[
\text{HDM}[T, R] \quad \text{andThen} \quad \text{HDM}[R, U] \Rightarrow \text{HDM}[T, U] \quad (5)
\]

- **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.

2) Composition of HDM applications: Let us consider the WordCount program (Fig. 1) as an example with the assumption that the program is already deployed on HDM, and a subsequent developer wants to write an extended program that counts the sum of all words that start with the letter ‘a’. In order to support this extension, 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 it. 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. 2). 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.

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

Figure 2. Apply new operation to an existing program

An additional example is that sometimes developers want to reuse the same program to process datasets from different sources with different formats. In HDM, developers can 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. 1) 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 just need to write a few lines of code as shown in Fig. 3.

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

C. Interaction with HDM

<table>
<thead>
<tr>
<th>Action</th>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>compute</td>
<td>References of computed HDMs.</td>
</tr>
<tr>
<td>sample</td>
<td>Iterator of a sampled sub-set for computed records.</td>
</tr>
<tr>
<td>count</td>
<td>Length of computed results.</td>
</tr>
<tr>
<td>traverse</td>
<td>Iterator of all computed records.</td>
</tr>
<tr>
<td>trace</td>
<td>Iterator of task information for last execution.</td>
</tr>
</tbody>
</table>

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 III), jobs are dynamically submitted to the related execution context which can be either a multi-core threading pool or a cluster of workers. Fig. 4 shows how to interact with the WordCount job and print out the output on the client side.

During execution, the composed function could be directly executed on every input block of data within one task.

IV. Data Flow Optimizations on HDM

During execution, HDM jobs are represented as functional DAG graphs, on which multiple optimizations could be applied to get better performance. In this section, we will take the WordCount job shown as an example to explain how optimizations are applied to a HDM job.

A. Parallel Function Fusion

In HDM, we define parallel operations as a sequence of operations that start with One-To-One or N-To-One data dependency and end with One-To-One or One-To-N data dependency. These parallel operations could be fused into one HDM rather than separated ones. Parallel functions 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 parallel fusion in HDM can be specified as:

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

B. Local Aggregation

Local aggregation is a very useful approach to reduce the communication cost for shuffle operations with aggregation semantics. For these 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 $f : T \to R$, 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) plus a followed by $HDM_g[R, U]$ with the aggregation semantics $F_a$ and original shuffle dependency.

As an example, let us consider the data flow which is shown in Fig. 5. By detecting the aggregation operation ReduceByKey after GroupBy, two 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. 6. After local aggregation, new parallel sequences are generated. Therefore, function fusion is triggered to combine parallel operations. Eventually, the final data flow is simplified as Fig. 7.
TABLE IV. REWRITING PATTERNS IN HDM

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Re-written</th>
</tr>
</thead>
<tbody>
<tr>
<td>map(m : T → R)</td>
<td>filter(f (m) : T → Boolean) .map(m)</td>
</tr>
<tr>
<td>intersection(f : T → Boolean)</td>
<td>filter(f : T → Boolean).intersection()</td>
</tr>
<tr>
<td>sort(f : T → Boolean)</td>
<td>filter(f : T → Boolean).sort()</td>
</tr>
<tr>
<td>groupByKey(f : K → T)</td>
<td>filter(g : T → Boolean) .groupBy(g)</td>
</tr>
<tr>
<td>reduceByKey()</td>
<td>filterByKey(f : K → Boolean) .reduceByKey()</td>
</tr>
</tbody>
</table>

C. Re-ordering/Re-construction Operations

Apart from aggregation operations, there is another set of operations (like Filter or FindByKey) 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 the optimizer attempts to lift these pruning-operations to reduce data size in advance while sinking the operations which involves global aggregations (such as global Reducer and GroupBy) 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-write 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 the WordCount program extended with two functions: FindByKey and ReduceByKeyValue, as shown in Fig. 7.

During optimization, 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 as indicated in Fig. 8.

V. 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.
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.

1) **Logical Planning:** 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. 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 discussed in Section [49].

2) **Physical Planning:** 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. Physical planning is a low-level explanation which splits the global HDM operations into parallel ones according to the parallelism and data locations.

3) **Job Scheduling and Execution:** In the scheduling phase, the HDM scheduler simply takes out the activated tasks for which all the input data are computed and available. Then, the scheduled tasks are assigned to workers that are closest to the input locations. The distances between workers and data set are calculated by the sum of weighted distance between the worker and all the data partitions (weights are allocated as the data size of each partition).

VI. Evaluation

A. 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. In the image-classification pipeline, components like feature extractor and classification trainer are commonly-used algorithms for many machine learning applications. So they could be developed by other developers and published as shared applications in HDM. Then, subsequent developers can find those exiting HDM descriptions then directly reuse and integrate them to create a simple image classification application by writing just a few lines of code as shown in Fig. 11.

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 [4], Pig [5] and Oozie [4] 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 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, HDM server is acting as both an execution engine and application repository which enables developers to easily check and integrate with published HDM applications.

B. Performance Evaluation

In this section, we show the results of comparing the performance of HDM with Apache Spark (version 1.1.1).

1) **Experimental setup:** Our experiments have been applied on Amazon EC2 with one M3.large instance as master and 20 M3.2xlarge instances as workers (8 vCPUs, 30GB memory, 1000Gib network). Both Spark and HDM are running in a memory-only model on the JVM with 24 GB memory. In addition, data compression options in Spark are all disabled to avoid performance interference. For testing datasets, we use the user-visits and page-ranking data sets from AmpLab benchmark. The user-visits data is 119.34 GB; the page ranking data set is replicated to 22.28 GB to fit into a larger cluster. Before running experiments, all tested data sets are downloaded into a HDFS (Hadoop 2.4.0) which is hosted on the same cluster for executing the tested jobs. During experiments, we compare both primitives and different types of pipelined operations for HDM and Spark. The test cases are categorized into two groups: basic primitives and pipelined operations as listed below (TC-i denotes the i-th test case).

**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.

```scala
/* 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 */
extraconr= 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(e) ⇒ println(e)
}
```

Figure 11. Creating an image classification pipeline in HDM

\[\text{http://incubator.apache.org/flume}\]
\[\text{https://aws.amazon.com/ec2/}\]
\[\text{https://amplab.cs.berkeley.edu/benchmark/}\]
• TC-3, Shuffle with map-side aggregation: A Reduce-ByKey operation which groups the page ranking according to the prefix of page and sums the total value of every group.

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-4, Parallel sequence: five sequentially connected Map operations each of which transforms the key of the input into a new format.
• TC-5, Parallel sequence followed by pruning: five sequentially connected Map operations as same as test case 1, then followed with a filter to extract $1/3$ of the data.
• 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 ranking starting with certain prefix.
• TC-8, Shuffle operation followed by transformation: a groupBy operation which groups the page ranking according to the prefix, then append with a map operation to transform the grouped results into new formats.

2) 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. 12 for the parallel operations like Map, HDM shows better performance than Spark for both tested data sets. The main reason is that HDM provides a global view in planning phases so workers are assigned with parallel branches in the execution graph rather than a single step. In this case, less communication is required for completing the same job. Besides, the Spark scheduler uses delay scheduling to achieve better data localization by default. But for HDM, data localization and fairness are pre-considered in the physical planning phase, so no delay scheduling is required at runtime, which also decreases the latency of scheduling to some extent.

Simple shuffle operation: For single GroupBy operation, HDM and Spark have similar performance as no optimizations can be applied. However, HDM shows shorter JCT for smaller data sets while for larger data sets the JCT of HDM and Spark is quite close. Actually, Spark is more efficient for large-scale shuffling due to more sophisticated design and implementation in network components while the performance of HDM mainly benefits from the parallel data loading phase before shuffling.

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 apply local aggregation when necessary. Eventually, for optimized shuffle operations, HDM still achieves better performance than Spark, which is mainly due to the better efficiency in the parallel phases.

• Comparison of Pipelined Operations

Multiple parallel operations: For multiple parallel operations, Spark wraps sequentially connected operations into one task, within which pipelined operations are executed one by one iteratively. Due to this optimization, the JCT of TC-4 for Spark does not increase much compared to the single operation in TC-1. 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, HDM shows slightly better performance than Spark for this group of test cases.

Shuffle operation followed by data transformation: HDM and Spark both show relatively longer JCT for shuffle operations followed with general transformations. 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 indicates shorter JCT for smaller data sets due to more efficient parallel phase, while Spark scales better to larger data sets due to more efficient network IO implementation.

Shuffle operation followed by aggregation: For operations composed by shuffle and aggregation, HDM shows much better performance 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, 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.

3) 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. The experiments in this section are not supposed to show which framework is essentially better but to show that a process engine can significantly benefit from optimizations of functional DAG flow in terms of Job Completion Time.

In our experiments, HDM shows worse scalability for shuffle-intensive operations in comparison with Spark because Spark has more sophisticated design and implementation for network IO and data storage. For example, optimized data structures such as CompactBuffer and AppendOnlyMap are used in Spark to improve the performance of shuffling.

VII. RELATED WORK

A. Big data processing frameworks

Several frameworks have been developed for provided distributed big data processing platforms. MapReduce 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. For fast and smaller jobs where the data can fit into memory, Map-Reduce framework is usually not effective.

Spark [9] utilizes memory as the major data storage during execution. Therefore, it can provide much better performance compared with jobs running on MapReduce. The fundamental programming abstraction of Spark is called Resilient Distributed Datasets (RDD) [10] which represents a logical collection of data partitioned across machines. Besides, application on Spark are decomposed as Stage-based DAGs that 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 re-ordering and rewriting are not provided in the Spark processing engine.

Apache Flink [7] 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 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. TABLE V compares the main features between HDM and major open-source big data frameworks.

### B. Optimizations for Big Data Applications

FlumeJava [2] is a java library that recently introduced by Google. It is built on MapReduce and provides a higher level wrapping and a bunch of optimization 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.

Tez [6] is a graph-based optimizer which can significantly optimize MapReduce jobs written in Pig and Hive. Basically, Tez simplifies the MapReduce pipelines by combining multiple redundant Mapper and reducers. Besides it 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.

### VIII. Conclusion and Future Work

In this paper, we have presented HDM as a functional and strongly-typed data representation, 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 execution 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.

However, HDM is still in the initial stage, while some limitations are left to be solved in our future work: 1) disk-based processing needs to be supported in case that the overall cluster memory is not sufficient 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 heterogeneous data sets, which normally cause heavy outliers and seriously slow down the overall job completion time and degrade the global resource utilization.

### REFERENCES


---

https://flink.apache.org/