2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing

Towards Big Data Analytics across Multiple Clusters

Dongyao Wu*†, Sherif Sakr*†‡, Liming Zhu*†, Huijun Wu*†

E-mail: {Dongyao.Wu, Sherif.Sakr, Liming.Zhu, Huijun.Wu}@data61.csiro.au
*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

***Abstract*—Big data are increasingly collected and stored in a highly distributed infrastructures due to the development of sensor network, cloud computing, IoT and mobile computing among many other emerging technologies. In practice, the majority of existing big-data-processing frameworks (e.g., Hadoop and Spark) are designed based on the single-cluster setup with the assumptions of centralized management and homogeneous connectivity which makes them sub-optimal and sometimes infeasible to apply for scenarios that require implementing data analytics jobs on highly distributed data sets (across racks, data centers or multi-organizations). In order to tackle this challenge, we present HDM-MC, a multi-cluster big data processing framework which is designed to enable the capability of performing large scale data analytics across multi-clusters with minimum extra overhead due to additional scheduling requirements. In this paper, we present the architecture and realization of the system. In addition, we evaluate the performance of our framework in comparison to other state-of-art single cluster big data processing frameworks.**

I. Introduction

The amount of digital data has been explosively increasing for the past decades. According to a recent report from IDC, by 2020 the total digital data size will be 300 times bigger than it was in 2005 and the amount of data is predicted to almost double every two years in the future [1]. Simultaneously, with the emergence and development of cloud computing, mobile computing and the Internet of things, data are increasingly being collected and stored in highly distributed infrastructures (e.g. across data centers, clusters, racks and nodes). During past years, many frameworks have been developed to deal with ever larger data sets on ever larger distributed clusters. The majority of these big-data-processing frameworks such as Hadoop and Spark [18] are natively designed and implemented based on the single-cluster design in which, there are two basic assumptions:

- *Centralized cluster management assumption*: which assumes the whole cluster is managed by a centralized master (sometimes with a standby backup) which is responsible for the resource management, job explanation and scheduling.
- *Homogeneous assumption*: which generally assumes that all nodes in the cluster are symmetrically connected and data are generally distributed among them.

However, these two assumptions do not fit into for many scenarios in which data are maintained in a highly distributed environment and there could be either physical or logical boundaries for various groups of computational nodes.

In principle, the data center infrastructure itself is becoming bigger and more complicated so that the connectivity difference between different sub-sets of nodes are not negligible. In practice, for widely distributed data sets, the connectivity including bandwidth and latency between different blocks/paritions of data can significantly differ. For example, transportation between nodes under the same rack can be very fast; transportation across different racks can be considerably slower; transportation across different geo-located data centers can be much slower and even be more limited. Omitting the heterogeneity of underlying infrastructure can result in significant performance degradation [6], [10], [14]. To address this issue, several research efforts have attempted to extend the MapReduce framework to support highly distributed environments such as Grid [2], [11] and multi-clusters/clouds [3], [5], [15]. As geo-distributed analytics (GDA) is becoming more important, there is also a family of works that extend MapReduce to apply query execution and optimizations on those geo-distributed data sets [10], [13], [14]. Recently, functional big data processing frameworks such as Spark [18] and Flink[1] have become the promising next generation of big data processing frameworks due to their declarative and data-oriented nature with richer programming interfaces. Apart from SQL queries, general analytics programs (such as machine learning and graph-based algorithms) are also very important for modern data analytics applications. However, such frameworks lack of sufficient support for general data analytics on highly distributed infrastructures.

In practice, data are increasingly distributed across different organizations that may hold different features for the same entities due to the diversified products they provide. As a result, there are increasing requirements to apply data analytics across organizations in order to discover more comprehensive patterns and knowledge for data scientists and end users. However, for such scenarios, current big

[1] https://flink.apache.org/

978-1-5090-6611-7/17 $31.00 © 2017 IEEE
DOI 10.1109/CCGRID.2017.73

data processing frameworks (e.g., MapReduce, Spark) are not natively designed to support coordination and computation across multi-clusters. Existing large scale cluster management frameworks (e.g, Yarn, Brog [12]) mainly rely on a centralized master to manage massive nodes/clusters, which are not suitable for the multi-party scenarios where multiple masters are required in different organizations due to regulatory and privacy concerns [9].

In order to tackle the above challenges, we present HDM-MC (HDM Multi-Cluster), a multi-cluster solution that enables the capabilities of performing large scale data analytics over multiple clusters for both single and multi-party scenarios. In addition, we describe a set of optimizations that have been introduced to improve the performance for multi-clustered applications. In particular, we summarize the main contribution of this paper as follows:

- We present two multi-cluster architectures (P2P and Hierarchical) which are designed to support managing and coordinating resources (workers) on multi-clusters infrastructures.
- We present a two-layer job planning and scheduling mechanisms which are designed to support the coordination and execution of big data jobs across multi-cluster infrastructures.
- We conduct comprehensive set of experiments to evaluate the performance of our multi-cluster framework in comparison to the state-of-art big data processing frameworks.

The remainder of the paper is organized as follows. Section II illustrates the motivations for our work. Section III presents the our multi-cluster architecture and its realization. Section IV presents the experimental evaluation about our multi-cluster solution. We discuss the related work in Section V before we conclude the paper in Section VI.

II. Motivating Examples

In this section, we highlight the limitations of applying a single-cluster big data processing framework on different scenarios and applications that motivate the need of our proposed framework.

A. Multi-clusters in one Organization

As the data centers have become bigger and more complicated then ever before, there are increasing requirements for constructing multi-clusters in one organization which has massive infrastructures:

- *Logically Separated Multi-clusters*. In current capital companies, they normally has multiple products each of which may have their databases and data repositories. Although, different products can share the same physical infrastructure for their data analytics clusters, different products still prefer to separate their own data sets and management of their computing resources because each product may has different data sensitivity and computation priority considerations. In this case, many large companies logically divide their data processing infrastructures for different product groups.
- *Geo-distributed Multi-clusters*. For global companies such as Google, Facebook and Amazon, they have customers all over the world. Therefore, multiple data centers are setup geographically in order to provide low-latency services to users in different regions. Respectively, there are multiple data processing clusters being constructed in each of this data centers. It is a great challenge for data scientists to apply data analytics over the data across all the data centers as there is very limited network connectivity across those geo-distributed clusters.

In practice, current big data processing frameworks (e.g., MapReduce, Spark, Flink) are originally designed for single cluster infrastructures. Therefore, there is crucial needs to provide multi-cluster solutions to tackle the above challenges and requirements.

B. Multi-party Computations across Organizations

As data are increasingly distributed across different organizations and different organizations may hold different features for the same entities due to the different products they have. There are increasing requirements to apply data analytics across organizations in order to discover more comprehensive patterns and knowledge from end users. For example, disease prediction and treatment based on biometric is becoming popular in recent years. In this scenario, DNA collecting companies hold the DNA information from many people while other hospitals hold the patients data, respectively. They are willing to apply analytics on each other's dataset to be able to get more accurate predictions and recommendations for people and patients but they are both not willing to share the raw data to others due to confidential and privacy reasons. There have been a lot of multi-party algorithms being able to apply analytics across datasets without disclosing the original contents. However, there are no mature and stable frameworks to support the ability to write and execute those multi-party algorithms in current big data ecosystems. Current big data processing frameworks such as MapReduce and Spark are not natively designed to support coordination and computation across multi-clusters and organizations.

Figure. 1 illustrates the problem of applying the Spark framework in multi-party scenario. In principle, Spark framework does not support coordination between different clusters within a single driver program. In addition, there is no mechanism to support locating data across multi-clusters in Spark during execution. Therefore, when a developer tries to write a algorithm for processing data from multiple clusters, there are a lot of data loss exceptions caused by the inability to recognize remote blocks in another cluster.

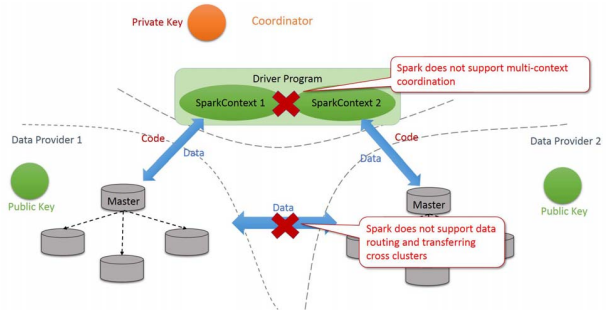


Figure 1. Multi-party Computation using Spark.

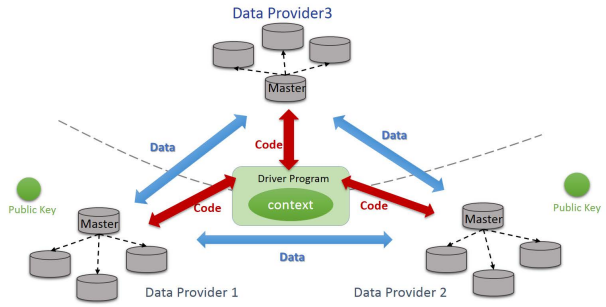


Figure 2. A Better Solution for a Multi-party Computation Architecture.

C. A Multi-cluster Solution

In order to fulfill the presented requirements, we present, HDM-MC, our multi-cluster solution to natively support the ability to perform computation and data analytics across multiple clusters with both physical boundary (structured regrading to the network topology) and logical boundary (structured regrading to multi-party/organization relation). Figure 2 illustrates an overview of our framework which is targeting the following three basic requirements:

- *Multi-cluster coordination and management*: Supporting the collaborative execution of data analytics jobs across multiple clusters, each of which manage their own computation and data resources without directly exposing them to other clusters.
- *Data transparency with localization*: Data are automatically routed and located within the multi-cluster infrastructure. External system and users should not consider and know about the data being exchanged within the cluster. On the other hand, the multi-cluster servers are able to provide optimized task planning and scheduling based on the awareness of the underlying network topology.
- *Simple and unified programming interface*: Users should be able to write one single driver program and algorithm which is sufficient to coordinate with multiple clusters without the need to consider the additional issues of data localization and exchange.

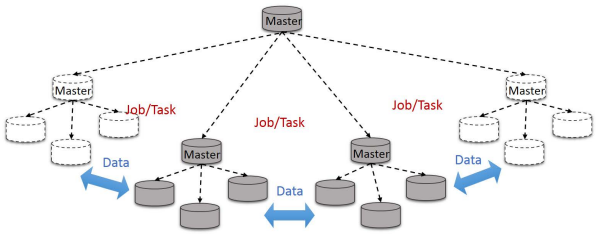


Figure 3. Hierarchical Multi-cluster Architecture.

III. HDM-MC: A Multi-Cluster Architecture

A. Core Execution Engine - HDM

HDM-MC is designed as an extension of our previous framework, the Hierarchically Distributed Data Matrix (HDM) [17]. In principle, HDM is a light-weight, functional and strongly-typed data representation which contains complete information (such as data format, locations, dependencies and functions between input and output) to support the parallel execution of data-driven applications. In order to enable HDM to support multi-cluster jobs, our extensions mainly include the following three main components:

- *Coordination of multi-clusters with two architectures*: Hierarchical supervisors and master chains and provide mechanisms to support dynamic switching between single and multi-cluster architectures.
- *Multi-cluster planning*: Stage-based planning by checking the context of the dependent data source and differentiate different jobs: Local Job, Remote Job and Collaborative Job.
- *Optimized task scheduling for multi-cluster*: Scheduling strategies with the awareness of the underlying network topology (distance between nodes).

B. Coordination of Multi-clusters

The first step for supporting multi-cluster applications is to provide support for coordination between different masters for each cluster. In our solution, we provide two types of coordination architecture considering different collaborating scenarios for applications: A hierarchical architecture and P2P architecture. The former is suitable for applications which allow an external coordinator over the masters of each cluster. The latter is suitable for clusters which only trust peer masters that they have recognized and agreed with. In the remaining of the section, we will illustrate the design of each architecture in detail.

1) Hierarchical Architecture: In the hierarchical architecture, there is one or more super-masters who are able to perform the coordination for the children clusters. Figure 3 illustrates an overview of the hierarchical multi-cluster architecture. From the super-master's point of view, it treats the children masters as normal workers each of which contains the resource by summarizing the total resources (e.g. CPU cores and memory) of the actual workers in each

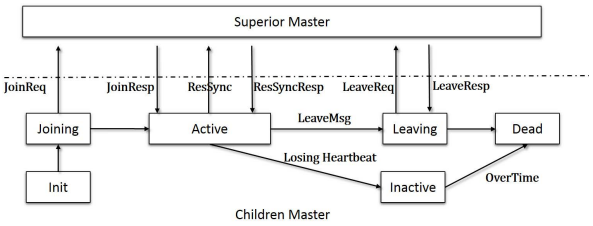


Figure 4. Message Coordination between Hierarchical Clusters.

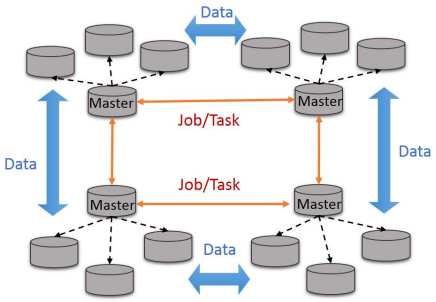


Figure 5. P2P Master Architecture.

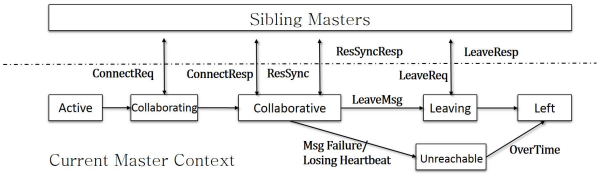


Figure 6. Message Coordination between P2P Clusters.

cluster. From the perspective of mediation masters, they are responsible to manage and monitor the resources of underlying workers and also keep reporting on the changes of the cluster (including itself) to the direct superior master. In principle, this architecture could be extended to an multi-layer (with multiple layers of mediation masters) hierarchy as the superior master actually considers the children masters just as normal but more "powerful" workers.

Figure 4 shows the states of the life cycle of a master and its message coordination between its superior master. A children master has six basic states:

- *Init*: When a master process is started, it is in a init state.
- *Joining*: When a children master is requesting the supervision from a superior master, it sends a JoinMsg (with the information of the resources in the current cluster) to the superior master then changes its state to Joining. When a children master is in the Joining state, it will postpone the offers which might cause resource changes to make sure the resource info is consistent as described in previous JoinMsg.
- *Active*: When a children master is in the Joining state, if it receives a successful JoinResp from the superior master, it will become Active and start to serve regular requests from both workers and its superior master. In the Active state, any operation that results in resource changes (including, worker joining/leaving and resource assigning/recycling) would trigger the children master to send the ResSync (Resource Synchronization) message to synchronize the changes to the superior master.
- *Inactive*: When a children master is in the Active state, any communication failure between its superior master will cause it to become Inactive. In the Inactive state, the children master will stop serving regular requests but keeps retrying the failed communication until it is succeed or time-out.
- *Leaving*: In the Active state, a children master can receive the LeaveMsg from either a client or superior master to actively leave the cluster. After successfully receiving the confirmation (LeaveResp) from the superior master, it will be changed to a Dead state.
- *Dead*: If a children master has been Inactive for too long or it has successfully left the cluster, it will be labeled as Dead. A dead children master will be permanently removed from the superior master.

2) Decentralized Architecture: Our framework supports a decentralized architecture which has no super masters, but instead, each master can have a few (e.g. two) sibling masters with which they can collaborate and share certain extent of computation and data resources. Figure 5 shows the overview about the chained multi-cluster architecture.

In this architecture, each master is responsible for managing its own worker children and updating resource info to its peer masters. During the execution of the applications, if some jobs are identified as remote jobs or require collaborative execution, the master can submit those jobs to the related siblings. To reduce the overhead for synchronizing resource information to sibling masters, the number of peer master are limited for each individual master (i.e two or three). Theoretically, this architecture is able to scale to a very large number of nodes with multiple peer-to-peer masters. Figure 6 shows the state diagram and message coordination between a master and its siblings with a few different states that are related to the coordination of the hierarchical architecture:

- *Collaborating*: Only masters which are in Active state can try to connect with other peer masters to start collaborating. Once a `ConnectReq` message is sent to a sibling master, the current master will change to a Collaborating state. Unlike the Joining state in hierarchical architecture, masters in the Collaborating state can still serve regular requests from its local cluster but will postpone all the requests from other peer masters.

- *Collaborative*: Once a Collaborating master receives a `ConnectResp` message from the required sibling, they will become collaborative to each other. Then, they start to synchronize the resource information whenever it changes. At the same time, they are able to send and receive remote jobs and tasks as well as the completion notification from each other.
- *Unreachable*: Any communication failure (heartbeat and other messages) between collaborative sibling masters will lead them to become Unreachable to each other. A Unreachable master would still be able to serve the requests from local cluster but will stop taking requests from its siblings unless it is reconnected.
- *Left*: If a sibling is unreachable over a time limit or it has actively left from the collaborating masters, it will be labeled as left. Left masters could still work separately within their own cluster.

3) Dynamic Architecture Switching: Both single and multi-cluster architectures have there own pros and cons. Single cluster is easier to construct and maintain while multi-cluster architecture suits better in some more complicated infrastructures. So it is meaningful to provide dynamic switching approach so that infrastructure managers is able to dynamically change their cluster architecture during infrastructure evolvement or supporting specific scenarios. In our work, we provide both mechanisms for a HDM cluster to switch its architecture between single and multi-cluster.

- *Single-cluster to multi-cluster*. To divide a single cluster into multiple clusters, the system admin can create a new master and migrate the workers from the old cluster to the new one. Then send a collaborating message to them. To migrate from one master to another, a worker just needs a LeaveMsg message to the current master. After the leaving message is confirmed, it sends a JoinMsg to the new master to complete the "migration" process.
- *Multi-cluster to single-cluster*. To combine a multi-cluster into a single cluster, the system admin can migrate all the workers to a selected master. Then, send leaving messages to other collaborating masters.

C. Job Planning and Scheduling of Multi-clusters

With the coordination service described in the last section, we are able to share the realtime resource information (including nodes, cpus, memory) of clusters between multiple masters. However, in order to support data analytics jobs across a multi-cluster architecture, the masters of the clusters need to recognize and explain data flow of the jobs and schedule the computation process on the cross-cluster resources. In this section, we present the job planing and scheduling processes of multi-cluster applications.

1) Categorizations of Jobs: Firstly, we categorize jobs into three basic categories according to the location of data sets and the point of job planning.

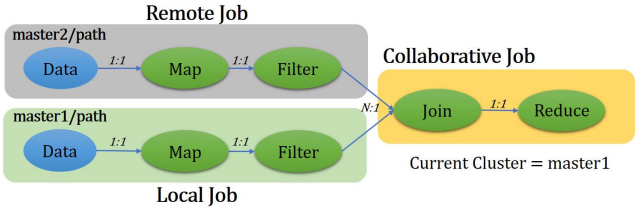


Figure 7. Job Explanation Example.

- *Local jobs*: If the input data sets of a job are all in the context of current cluster which performs the job planning, it is considered as a local job. For this type of job, it is scheduled as a normal local job on the children workers of the current cluster.
- *Remote jobs*: If the input data sets of a job are all in the context of another remote cluster, it is considered as a remote job. For a remote job, it is directly re-submitted to the related cluster for execution, meanwhile, a promise entry is registered in the current scheduling cluster to wait for the call-back response for the remote job after it has succeeded or failed.
- *Collaborative jobs*: If the input data sets of a job are from multiple different clusters (siblings), it is considered as a collaborative job. A collaborative job will be parallelized and scheduled on the overall resources of both current cluster and siblings regarding the data location and parallelism of the job.

Job categorization enables the planner and scheduler to differentiate jobs with different data sources from different clusters and locations. In addition, it also provides a clue for dataflow construction in the job explaining phase.

2) Job Explanation: After an analytics application is submitted to any of the master, it will be firstly explained as an understandable task/job flow for the scheduler to schedule the tasks/jobs. Based on the granularity and abstraction levels of jobs, the job explaining phase contains two steps: Stage Planing, and Task Planning.

- *Stage Planing step*: In this step, the application will be divided into several jobs stages, each of which belongs to one of the job categories.
- *Task Planning step*: For each job identified in the stage planning, it will be scheduled in one of the masters for execution. Before the actual execution, each master that receives the job stage will further explain the job into actual task flows for scheduling.

Stage Planning Step Algorithm 1 shows the process of identifying different types of jobs in the stage planning step. Basically, the stage planner recursively checks whether the children of a function belongs to the current cluster (context), if not the planner sets the current job stage as *Collaborative* and then creates new stages for each children based on their context.

Algorithm 1: StagePlaning

```
Data: current ctx of explaining, current node cN, current stage cS,
      parallelism p
Result: a list of job stages list_s identified
begin
    if children of cN is not empty then
        if cN.children are not all in ctx then
            cS.setType("collaborative");
            for each c in cN.children do
                if c.context == ctx then
                    nS := newStage(c, "local");
                    list_s += nS ;
                    cS.parents += nS;
                    list_s += StagePlaning(ctx, c, nS, p) ;
                else
                    nS := newStage(c, "remote");
                    list_s += nS;
                    cS.parents += nS ;
                end
            end
        else
            for each c in cN.children do
                list_s += StagePlaning(ctx, c, cS, p);
            end
        end
        return list_s;
    else
        return list_s;
    end
end
```

Figure 7 shows an example of the stage planning results for a cross-cluster program. In the program, the new stage is discovered at the *Join* node, for which the children comes from different cluster contexts. Then the two children of *Join* is categorized as *Remote Job* and *Local Job* while the *Join* node is categorized as *Collaborative Job*, respectively.

Task Planning Step After the job is divided into stages according to the context and data dependency, each divided stage is actually a self-executable job. Then each master just need to use the local job planner in HDM [17] to explain and schedule the local jobs. For collaborative jobs, it will wait for all the dependent stages to finish, then it is scheduled on the resources of either the current cluster or current cluster plus its siblings to improve the parallelism.

D. Scheduling on Multi-clusters

1) Multi-layer Scheduler Design: After the job is explained, it is required to schedule different types of jobs and coordinate with remote clusters to complete the tasks of the overall applications. Similar to the two-layer planning, our scheduling process is also designed as a two-layer scheduler.

- The first layer is responsible for monitoring and scheduling the stages of each application. Once all the parent jobs of a stage are completed, the stage is notified as active and will be submitted to a remote or local task scheduler for execution
- The second layer is responsible for receiving, monitoring and scheduling the tasks of each active stage. In principle, any classic single cluster scheduler can be re-used in this layer.

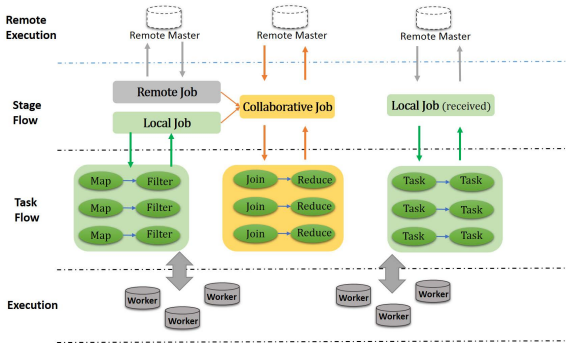


Figure 8. Multi-layer Job Scheduling Example.

Figure 8 illustrates the process of multi-cluster scheduling for previous example (Figure 7) from the perspective of one master. Basically, after a cross-cluster program/application is submitted to the server, it will firstly be divided into a stage flow based on Algorithm 1. During scheduling of the stage flow, if a stage does not have parents or all the parental stages have been completed, then it will be triggered and submitted to a local task scheduler or a remote siblings for execution. Based on the job type, a remote job will be submitted to the related sibling master and a local job will be submitted to the local task scheduler. A collaborative job will be explained in the current task scheduler and scheduled across the current cluster and its siblings (some task will be executed in local while some task will be submitted to a sibling master based on the data localization). Once the task scheduler receives a local or remote stage job, the job will be explained as executable task flows and then it is scheduled and executed as a local cluster job.

2) Scheduling Strategies: In the second layer of scheduler, explained tasks are scheduled in each local cluster. Currently, in our implementation, there are three types of scheduling strategies can be chosen:

- *Delay Scheduling*. A simple and commonly used strategy that allows arriving tasks to wait for a short duration of time in order to achieve better data locality. There are four main data locality levels in delay scheduling: Process Local, Node Local, Rack Local and Any.
- *Minmin/Maxmin Scheduling*. Tasks are scheduled based on the estimated minimum completion time. a) Firstly, the scheduler finds out the estimated minimum completion time for every task among all the resources; b) Secondly, the task with the minimum value within the candidate set is selected for execution.
- *Hungarian Algorithm*. It is originally a graph algorithm which is used find our the near optimal shortest distances among the nodes of a graph. In scheduling scenario, the distance matrix is calculated by the distance between input data and available candidate workers.

Delay scheduling has the least complexity among the three algorithms and it works well for non-shuffle operations. However, for shuffle operations, delay scheduling does not provide any optimizations and just randomly assign tasks to available worker. So it does not suit well for the multi-cluster scenario as shuffle operations are very expensive and network connectivity is heterogeneous. Both Minmin/-Maxmin and Hungarian algorithm are able to find more optimal scheduling plans for both shuffle and non-shuffle dependencies. However, they all have higher complexities: O($M \times N$) (M is the number of workers and N is the number if tasks) and O(N^3) respectively. During our experiments we choose Minmin Scheduling as our default scheduling strategy as it has higher scalability. In order to enable the awareness of heterogeneous network in multi-cluster infrastructure, the estimated execution time (T) and distance between worker and input data is calculated as the sum of CPU time (T_c), disk IO time (T_d) and network IO time (T_n).

$$T = T_c + T_d + T_n = d \times f_c + d \times f_d + d \times f_n \quad (1)$$

Each time component is calculated by the multiplication of input data size d with the related resource factor. By default, we consider the cpu factor $f_c = 1.0$, disk IO factor $f_d = 5.0$ and network IO factor $f_n = 15.0$ during estimation. These factors can be configured to fit into different execution environments in practice.

IV. Experimental Evaluation

We conducted comprehensive experiments to assess the different aspects of our framework using two group of experiments:

- We compare the performance of both multi-cluster architecture with the native single-cluster architecture on one local cluster infrastructure to find out the overhead for having a multi-cluster. For this group of experiments, we run our benchmark tests for native spark (single-cluster), single-cluster HDM and multi-cluster HDM on a EC2 cluster with 20 nodes.
- We compare the performance of multi-cluster architecture (HDM-MC) with the native single-cluster architectures (Spark and HDM) on a cluster infrastructure with limited inter-connectivity to find out the effects for using a multi-cluster in a heterogeneous environment. For this group of experiments, we run our benchmark tests for native Spark (single-cluster), single-cluster HDM and multi-cluster HDM on a infrastructure which contains two VPC clusters EC2 cluster (each with 10 nodes). The physical inter connectivity boundary between the two VPC is maximum of 1 Gb.

A. Experimental Setup

Our experiments are built on Amazon EC2 with up to 20 M3.2xlarge (8 vCPUs, 30GB memory, 1GB network) instances as workers and a variable number of M3.large instances for masters. Every Spark and HDM worker is running in the memory-only model on the JVM with 24 GB memory (480 GB aggregated memory for the entire cluster).

B. Benchmark and Test Cases

To better understand the performance of our multi-cluster realization for different types of applications, we run four groups of test cases for each type of cluster setting listed in section IV-A during our experiments. The four group of test cases include: basic primitives, pipelined operations, SQL queries and Machine Learning algorithms 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;
- 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 with 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 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*: Scans the data to find out records that meet the condition expression;
- TC-11, *OrderBy*: Orders the input data set by given column.

- TC-12, *Aggregation with groupBy*: Groups the records and aggregates the value of a numeric column in each group;

C. Experimental Results

During the experiment, we compared the Job Completion Time (JCT) for every tested case under each cluster setting. The results of each tested group are listed as follows.

1) Scheduling Overhead of Multi-cluster Architecture: We compare the scheduling cost (time spent in planner and scheduler) of multi-cluster architecture (HDM-MC) with the native single-cluster architecture (Spark and HDM) on a local cluster infrastructure to find out the overhead of introducing our multi-cluster architecture. Figure 9 shows the scheduling time (in ms) for scheduling of different types of test cases on the single-HDM cluster and Multi-HDM cluster. As we can see from the results, in general, the overhead costs of a more complicated scheduler in multi-cluster (HDM-MC) is generally less than 500 ms (avg. 355 ms). The average overhead across all the test cases for the for using HDM-MC is about extra 11.41% compared with the single-cluster scheduler, which is a acceptable time cost considering that most of jobs, the processing time can take at least few minutes to complete. The main cost of the multi-cluster scheduler comes from the two-layered scheduling in HDM-MC. As we mentioned in Section III-C2 and Section III-D1, HDM-MC contains the additional stage planning and stage scheduling steps at runtime to be able to explain and coordinate jobs which are executing across multiple domain and clusters. Sequentially, those extra steps induce additional cost during job planning and scheduling phases compared to the single-cluster scheduler which only needs to schedule local tasks.

2) Comparison of Performance on Heterogeneous Infrastructure: We compare the job completion time and data transfer of multi-cluster architecture with the single-cluster architecture (both Spark and HDM) on a two-cluster (each cluster has 10 nodes) infrastructure with limited interconnectivity to find out the impact of our multi-cluster solution in a heterogeneous environment.

For parallel operations such as Map (TC-1), MultiMap (TC-5), Select (TC-9), and Where (TC-10), Both Spark, HDM and HDM-MC shows similarly close Job completion time. The schedulers in Spark (Delay Scheduling) HDM (Minmin Scheduling for both single and multi-cluster architecture) are able to achieve data locality for the majority of the input for those parallel test cases. Therefore, almost no across cluster communication is needed during processing as shown in Fig. 11.

For shuffle-intensive test cases including GroupBy (TC-2), Sort (TC-4), GroupBy-with-Transformation (TC-6) and OrderBy (TC-11), there is no optimizations for spark while HDM and HDM-MC is able to achieve more optimal data locality by estimate the computation cost in scheduler based on to the input data distribution in the shuffle stage. Therefore, HDM shows shorter job completion time due to less data is transferred (as shown in Fig. 11) during shuffling.

For aggregation based shuffling, the performance is close for all the tested platforms. Much less data transfer across cluster is required for this type of jobs as shown in Fig. 11. Spark applies Map-Side merge in primitives such as ReduceByKey (TC-2) and uses Catalyst to optimize Aggregations (TC-12) in SQL. In HDM and HDM-MC, the optimization is achieved by Local Aggregation [17].

For pipelined operations with aggregation (TC-7) and filter (TC-8), HDM is able to apply operation reconstruction and reordering, which significantly reduces the data transfer during shuffling (as shown in Fig. 11). However, Spark does not provide any optimizations for these types of jobs. Consequently, HDM shows much better performance for this group of test cases, especially under a multi-cluster infrastructure. In addition, HDM-MC achieves the multi-cluster feasibility with the trade-off of only slightly longer job completion time compared with the HDM single cluster architecture. The main cost comes from the related longer scheduling time in the two-stage planing and scheduling in multi-cluster architecture plus the less-optimal overall execution plan as each master only hold half of the information for the workers compared to single-cluster solution which hold all the information in a global view.

3) Discussion and Conclusion: In our evaluation and experiments, we show that our solution is able to achieve the feasibility of construct a multi-cluster architecture with only losing small extent of scheduling cost and achieves very close performance compared to the optimized single cluster infrastructure of HDM. In addition, with our multi-cluster solution, it is able to apply data processing and analytics for both the multi-party scenario and the multi-cluster infrastructure in single organization. We also provide the flexibility of switching between single and multiple cluster architecture (as presented in Section. III-B3) to avoid unnecessary performance lost when it is not needed.

V. Related Work

In principle, the MapReduce framework is originally designed to operate on single-cluster environments. Therefore, it is not well developed to support the execution on highly distributed infrastructures and widely-networked clusters. To address this issue, many research works have attempted to extend the MapReduce framework to support highly distributed environments such as Grid [2], [11], multi-clusters/clouds [3], [5], [15]. In particular, Luo, et al. [8] present a Hierarchical MapReduce framework that introduces global reduce and locality-aware scheduling. They present another hierarchical framework [7] which can coordinate multiple clusters to run MapReduce jobs among them. Hung, et al Proposed the Workload-Aware Greedy Scheduling (SWAG) algorithm [4] which coordinates job scheduling across data centers with low overhead while

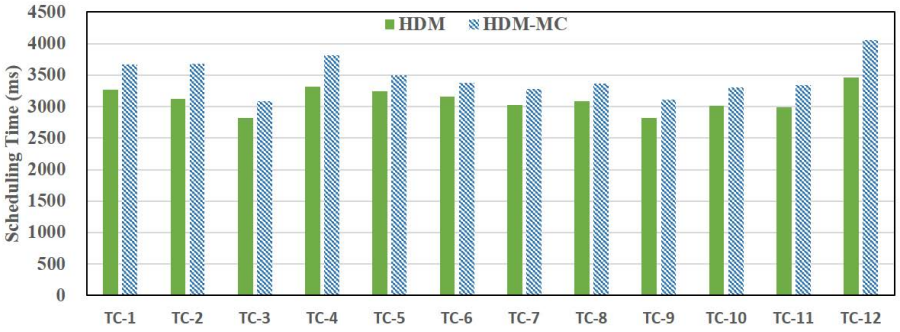
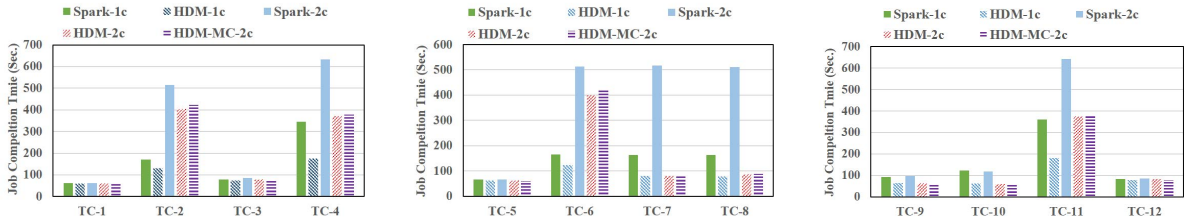
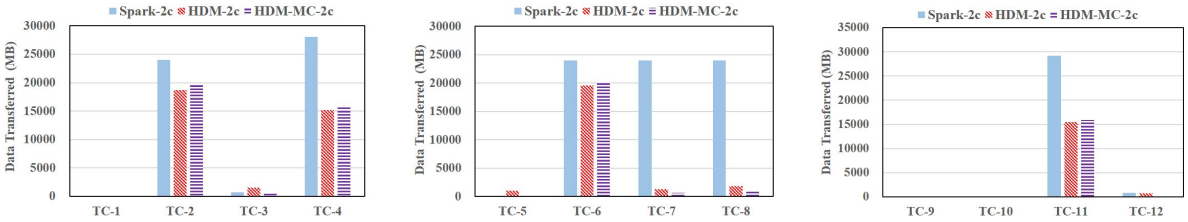


Figure 9. Comparison the scheduling cost of single and multi-cluster architecture.



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

Figure 10. Comparison of Job Completion Time on a two-cluster Infrastructure
Note: Postfix `-1c` and `-2c` denotes the results on a single cluster and two-cluster infrastructures respectively.



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

Figure 11. Comparison of Data Transfer on a two-cluster Infrastructure

achieve near optimal performance. G-Hadoop [16] enables Hadoop to support schedule data processing on multi-data centers/clusters and can provide larger pool of processing and data storage. Jayalath, et al. present G-MR [5], a MapReduce framework that can efficiently execute MapReduce jobs on geo-distributed data sets. G-MR optimizes data movement by finding the shortest path in DTG graph. However, the approach is highly complex and does not support complicated job sequences well. Compared with this group of works which focus on extending MapReduce to support highly distributed environment and geo-distributed data sets, HDM-MC provides the capability to explain and schedule general functional data analytics applications on multi-cluster infrastructures. A comparison between HDM and the key related works is listed in Table I.

As SQL based queries are one of the most commonly used interfaces for data processing and analytics, there are a group of works try to support geo-distributed queries for highly distributed environments. CLARINET [13] optimizes the query execution plans with the consideration of WAN-awareness to improve the performance of SQL queries on geo-distributed analytics. Iridium [10] achieves low query response time for query execution by optimizing placement of both tasks and data. Iridium uses online heuristic to redistribute data sets among sites prior to query evaluations. Vulimiri, et al.implemented a prototype Geode, schedules and optimizes query plans and data replications to improves the SQL analytics over geographically distributed data sets [14]. While those works focus on SQL query optimizations, general data analytics programs (such as machine learning and graph based algorithms) can hardly benefit from these optimizations. In HDM-MC, the job explanation and scheduling are applied on general programming primitives. Therefore, all the programming interfaces built on top of the

Table I. Comparison of HDM-MC and major related works.

	Execution Model	Optimizations	Cluster Architecture	Network Awareness	Multi-cluster
Hog [2]	MapReduce	Extend MR to Grid	Centralized	No	No
Ussop [11]	MapReduce	Extend MR to Grid Location-aware scheduling	Centralized Centralized	Yes Yes	No No
G-Hadoop [16]	MapReduce	Scheduler	Centralized	Yes	No
Iridium [10]	MapReduce	Data and Task Placement	Centralized	Yes	No
Geode [14]	SQL queries (Hive)	Query Optimizations	Centralized	Yes	No
CLARINET [13]	SQL queries (Hive)	Query Planning	Centralized	Yes	No
SWAG [14]	MapReduce	Scheduling	Centralized	Yes	No
G-Hadoop [16]	MapReduce	Scheduler	Centralized	No	No
G-MR [5]	MapReduce	data flow analysis	Centralized	Yes	No
HDM-MC	**Functional (HDM)**	**Data flow planning Scheduling**	**Hierarchical or Decentrlaized**	**Yes**	**Yes**

framework can take the benefits.

In terms of multi-cluster management, Google's introduced Borg [12] which is a centralized cluster manager that can manage a number of clusters each with up to tens of thousands of machines. HDM-MC provides similar cluster management capability while provides both centralized (hierarchical) and decentralized (P2P) architectures so that cluster operators can flexibly choose the architecture based on their own scenarios.

VI. Conclusion and Future Work

In this paper, we presented our work about multi-cluster solution, which fills the gap that when multi-cluster architecture is needed for complicated and highly distributed infrastructure. Our solution enables the multi-cluster feasibility with minimum cost in scheduling and with the optimizations in scheduling, the multi-cluster architecture shows reasonable good performance in our experiments compared with the state-of-art single cluster platforms. In addition, we also provide mechanisms to support dynamic switching of architecture between single and multi-clusters, which provide both convenience and flexibility for system administrators to dynamically evolve and manage their clusters based on different requirements.

Our work in this paper is the initial but important step to provide a thorough solution that supports data processing on complicated and heterogeneous infrastructures. There are many further work can be applied and extended in our future plan. Firstly, it is a challenging and promising work to find out data replication strategies to reduce the data transfers across network boundaries. Second, many data flow optimizations in single cluster infrastructure can be further explored under the multi-cluster scenarios. Last but not the least, security and trust models are crucial to make the solution applicable to realistic applications and products.

References

[1] P. Botteri. Eastern european champions and the 4 v's of big data.

[2] C. He, D. Weitzel, D. Swanson, and Y. Lu. Hog: Distributed hadoop mapreduce on the grid. In *SC*, 2012.

[3] B. Heintz et al. Optimizing mapreduce for highly distributed environments. *arXiv:1207.7055*, 2012.

[4] C.-C. Hung, L. Golubchik, and M. Yu. Scheduling jobs across geo-distributed datacenters. In *ACM SoCC*, 2015.

[5] C. Jayalath, J. Stephen, and P. Eugster. From the cloud to the atmosphere: running mapreduce across data centers. *IEEE Transactions on Computers*, 63(1), 2014.

[6] J. Jiang et al. Via: Improving internet telephony call quality using predictive relay selection. In *ACM SIGCOMM*, 2016.

[7] Y. Luo et al. A hierarchical framework for cross-domain mapreduce execution. In *Proceedings of the second international workshop on Emerging computational methods for the life sciences*, 2011.

[8] Y. Luo and B. Plale. Hierarchical mapreduce programming model and scheduling algorithms. In *CCGrid*, 2012.

[9] P. Mohan et al. Gupt: privacy preserving data analysis made easy. In *SIGMOD*, 2012.

[10] Q. Pu et al. Low latency geo-distributed data analytics. *ACM SIGCOMM Computer Communication Review*, 45(4), 2015.

[11] Y.-L. Su et al. Variable-sized map and locality-aware reduce on public-resource grids. *FGCS*, 27(6), 2011.

[12] A. Verma et al. Large-scale cluster management at google with borg. In *EuroSys*, 2015.

[13] R. Viswanathan et al. Clarinet: Wan-aware optimization for analytics queries. In *OSDI*, 2016.

[14] A. Vulimiri et al. Global analytics in the face of bandwidth and regulatory constraints. In *NSDI*, 2015.

[15] L. Wang et al. Mapreduce across distributed clusters for data-intensive applications. In *IPDPS Workshops*, 2012.

[16] L. Wang et al. G-hadoop: Mapreduce across distributed data centers for data-intensive computing. *FGCS*, 29(3), 2013.

[17] D. Wu, S. Sakr, L. Zhu, and Q. Lu. Composable and efficient functional big data processing framework. In *IEEE International Conference on Big Data*, 2015.

[18] M. Zaharia et al. Spark: Cluster Computing with Working Sets. In *HotCloud*, 2010.