Firebird: Network-aware Task Scheduling for Spark Using SDNs

Xin He, Prashant Shenoy
University of Massachusetts Amherst
{xhe, shenoy}@cs.umass.edu

***Abstract*—Recently Spark has become a popular cluster computing platform because of its fast in-memory computing which allows users to cache data in servers' memory and query it repeatedly. However, since network I/O is much slower than local I/O, the network can be a bottleneck for data intensive jobs. When data locality is not well balanced or network capacity is limited, data contention can occur, which will significantly slow down task execution. The current delay scheduling method in Spark cannot perfectly solve this problem because it is agnostic to the network status in a cluster. In this paper, we propose a network-aware scheduling method in Spark and design Firebird, a derivative of Spark that runs on top of software-defined network (SDN). By using SDNs, the communication barriers between cluster computing platform and underlying networking are removed and tasks can be scheduled based on network conditions in Spark clusters. We demonstrate the effectiveness of the methods through detailed experiments with different types of jobs on our system. Experimental results show significant improvement of data-intensive jobs, and our system can achieve best scheduling in different cases without tuning Spark. Firebird can be up to 9 times faster than default Spark in some cases.**

I. Introduction

Running analytic queries on large, diverse, and ever-growing datasets, also referred to as big data processing, has become an essential part of business processes for enterprises. MapReduce [8] has emerged as a framework for processing large amounts of structured and unstructured data in parallel across a large number of machines, in a reliable and fault-tolerant manner. Hadoop and Spark are two popular platforms that implement MapReduce framework.

MapReduce platforms provide computation, storage and network resources to users and efficient management of these resources is essential for platform performance. Current resource managers like Yarn and Mesos mainly focus on managing computing and memory resources in a cluster. Recent research has found that for data intensive jobs, network and disk I/O are major performance bottlenecks in MapRedce platforms like Hadoop [11]. But these aspects are not handled by current resource managers.

As server memory grows in size, it has been feasible to use persistent memory structure to address the issue of slow disk I/O. Resilient Distributed Datasets (RDDs) have been proposed to provide high efficiency data reuse which reduces disk I/O by using in-memory data storage. RDD is implemented in Apache Spark which performs up to 100 times faster than Hadoop [18].

However, due to the distributed nature of MapReduce framework, the network I/O has always been a scarce resource that limits the MapReduce's performance [1]. Neither Spark nor Hadoop explicitly manage network resources. Unbalanced data locality may cause biased network utilization which means data contention can happen and harm a cluster's performance during job execution [2]. Moreover, this problem becomes even more challenging when the network is shared with other applications [14].

The main cause of the problem is the separation between the current resource manager and networking i.e., resource manager is agnostic to the underlying networking while networking neglects any MapReduce-specific customization requests. As a result, if there is a node with adequate computing resource, but limited network bandwidth, networks become the bottleneck when many non-local tasks are scheduled to and executed on this node. In traditional MapReduce frameworks like Hadoop, this problem is not severe because local disk I/O is slow. But in Spark, in-memory data storage accelerates local data I/O so that network I/O is more likely to be the bottleneck and network congestion will largely impact the performance of Spark jobs.

In this context, we focus on the problem of how we can create a more collaborative relationship between Spark and networking in order to improve the performance by exploiting the capabilities offered by software-defined networking (SDN) [12], [13].

It is desirable to create a more collaborative relationship between Spark query processing and the underlying networking where there is a more direct and continuous communication between those two components for improved data processing. It is intuitive that if Spark has more visibility to the state of the network, it can make better decisions on scheduling the tasks across the distributed nodes. The problem is that, as pointed out earlier, data processing and network are separate entities that do not directly communicate with each other and network is generally managed distributedly which makes it hard to create the relationship between network and Spark. However, the emerging software-defined networking (SDN) is removing those communication barriers by providing direct APIs for applications to monitor and control the state of the network. Our goal in this paper is to explore SDN to create an efficient collaborative distributed query processing environment with Spark.

In this paper, we analyze scheduling problems of existing task schedulers in MapReduce frameworks and then propose a network-aware scheduling method which optimizes task scheduling based on network status of a cluster. We model different scheduling methods and analyze their performance under unbalanced data locality. Then we implement our method in Spark and build an actual system running on top of software-defined networking. In contrast to previous work, our work aims at building a collaborative relationship between Spark and a SDN-based network.

Our contributions: Our specific contributions over the existing work are the following:

(1) We present a network-aware scheduling method and show that by avoiding data contention, it improves Spark's performance under unbalanced data locality and unbalanced network utilization.

(2) We implement our scheduling method in Spark running on a "real" software-defined networking with OpenFlow enabled switches.

(3) Our experimental results clearly show the benefits of collaboration go beyond current scheduling methods and can achieve up to 9 times better performance than default scheduler in a shared network. Further, our system can always achieve best performance without tuning Spark when executing different types of jobs.

The rest of this paper is organized as follows. Section II presents background on Apache Spark and SDN. Section III presents the algorithm of network-aware task scheduler and analyses its performance in theory. Section IV presents the design of our system. Section V presents our experimental results. We present related work in Sec VI and conclude in Sec VII.

II. Background

In this section, we describe Spark and SDNs by way of background for subsequent sections.

A. Spark Background

Apache Spark is a MapReduce-like cluster computing framework that can execute data-parallel tasks. When a Spark job is submitted, the Spark master divides it into stages according to shuffle operations (one stage per shuffle operation). Figure 1 shows how one Spark stage is executed. The input data is RDD partitions stored on disk or in memory. Each RDD partition is corresponding to one map task in Spark. Map tasks are assigned to executors by Spark master and processed into key-value pairs. Then the shuffle operation deals with values for each unique key and generates output.

In contrast to traditional MapReduce engines, Spark introduces a distributed memory abstraction called Resilient Distributed Dataset (RDD). Each RDD is a read-only data structure created from data in stable storage or through a transformation on existing RDDs. Each RDD is divided into partitions and can be stored in memory or disk on individual servers. Users can explicitly cache RDDs in memory by using persist() or cache() function in Spark API. When servers have enough memory, a lot of data can be kept in memory which makes Spark much faster for jobs that need to iteratively read the same dataset, e.g. machine learning jobs and SQL queries. Experimental results show that Spark can be up to 100 times faster than Hadoop for specific jobs [18]. To deal with data lost during failures, Spark uses a directed acyclic graph (DAG) to record transformations that created the RDD. Importantly, transformations are coarse-grained in that they apply the same operation to each of an RDD's partition in parallel. Thus, if a node fail, the RDD partitions on this node can be re-generated on other nodes.

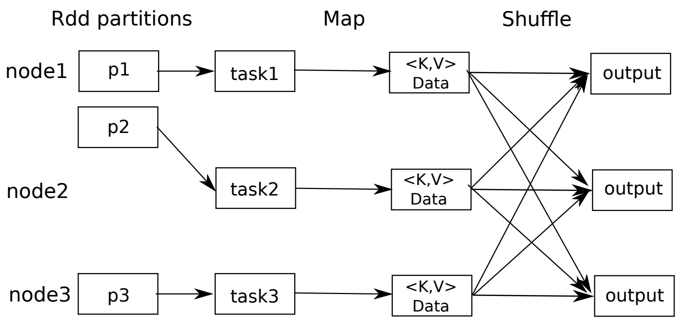


Fig. 1. Execution of a Spark stage

If the distribution of RDD partitions are not well balanced among nodes, e.g. one node has all RDD partitions or one node has no RDD partitions, there can be data shipping between nodes during execution (in Fig 1 p2 is shipped from node1 to node2). Data distribution of Spark cluster may be unbalanced for two reasons: 1) the data distribution in stable storage such as HDFS is not balanced 2) some nodes lose data because of node failures. To deal with this unbalanced data distribution, task scheduler should be able to judiciously schedule tasks to make full use of computation resource of the cluster as well as reduce latency caused by data shipping. Similar to Hadoop, Spark uses a locality-aware scheduler which divides tasks into 5 different locality levels and schedule tasks with high locality first. The five locality groups are:

PROCESS_LOCAL: the data is in the same JVM as the current executor. Data in this level is already deserialized and cached in memory, so it's very fast.

NODE_LOCAL: the data is on the same server as the current executor. Data can either be in local disk or HDFS directory on the node. If data has been used recently, it may also be cached in memory by OS file caching.

NO_PREF: the data has no locality preference which means data is not on the same node as the current executor and it's also not on other nodes that have executors. One example is that data is stored in Amazon S3.

RACK_LOCAL: the data is not stored on the same node as the current executor but on the same rack as the current executor. This level assumes the data shipping within a rack is faster than that between different racks.

ANY: the data is neither on the same node nor on the same rack as the current executor. Different from NO_PREF, data has its locality preference which generally means the node who owns the data can also execute this task. Therefore, this task may potentially be executed as PROCESS_LOCAL or NODE_LOCAL if it's not scheduled to the current executor.

For simplicity, in this paper, we treat PROCESS_LOCAL and NODE_LOCAL as "local" and the other 3 levels as "non-local". To monitor the network status of the whole cluster, we assume input data is stored in the cluster nodes rather than file systems like Amazon S3.

B. Software-defined networking and OpenFlow

SDN is a new approach to networking that decouples the control plane from the data plane. The control plane is responsible for making decisions about where traffic is sent and the data plane forwards traffic to the selected destination. This separation allows network administrators and application programs to manage network services through abstraction of lower level functionality by using software APIs [13]. From Spark's point of view, the abstraction and the control APIs allow it to (1) monitor the current status and performance of the network, and (2) modify the network with directives, for example, setting the forwarding path for non-local tasks.

OpenFlow is a standard interface among the layers of an SDN architecture, which can be thought of an enabler for SDN [12]. An OpenFlow controller communicates with an OpenFlow switch. An OpenFlow switch maintains a flow table, with each entry defining a flow as a certain set of packets by matching on 10 tuple packet information as shown below.

$$Flow ::= [InPort, VLANID, MACSrc, MACDst, MACType, IPSrc, IPDst, IPProto, PortSrc, PortDst]$$

When a new flow arrives, according to OpenFlow protocol, a "PacketIn" message is sent from the switch to the controller. The first packet of the flow is delivered to the controller. The controller looks into the 10 tuple packet information, determines the *egress port* (the exiting port) and sends "FlowMod" message to the switch to modify a switch flow table. When an existing flow times out, according to OpenFlow protocol, a "FlowRemoved" message is delivered from the switch to the controller to indicate that a flow has been removed. The controller can also send a "PortStatus" message to switches to get statistics like TX data and RX data. By periodically enquiring port status of all switches, controller can compute current network utilization and available bandwidth of all links in the cluster.

III. Task Scheduling

In MapReduce platforms like Hadoop and Spark, data may not be on the same node as executor, resulting in network traffic between nodes. For reduce tasks, since each task reads roughly equal amounts of data from all nodes [17], data locality based scheduling doesn't provide much benefit. However, since each map task is executing data on a specific node, if data locality of a cluster is not well balanced, map tasks can cause network congestion which may harm the performance of whole system. So in this paper, we mainly focus on scheduling of map tasks. In this section, we'll first introduce current scheduling methods and describe the problems of these schedulers. Then we propose our network-aware scheduling method based on knowledge of network status of a cluster. Finally, we create models of these scheduling methods to analyse their performance under unbalanced data locality.

A. Naive Scheduling

Naive scheduling was implemented in early versions of Hadoop. In naive scheduling, tasks are divided into different locality sets based on distance to the executor. When the scheduler receives a heartbeat from an idle node, it searches for available tasks from node-local set first, then rack-local set and at last non-local set. It does so because it believes that closer tasks can have better data transfer rate. Algorithm 1 shows the pseudo code of naive task scheduling.

Algorithm 1: Naive task Scheduling

```
Input: Node $n$ that has an idle slot
Output: Task $t$ to be assigned to $n$
for $j$ in $jobs$ do
    if $j$ has unassigned task $t$ with data on $n$ then
        return t;
    end
    else if $j$ has unassigned task $t$ with data on nodes in the same rack with $n$ then
        return t;
    end
    else if $j$ has unassigned task $t$ then
        return t;
    end
end
return null;
```

There are several drawbacks in naive scheduling. The first one is that whenever the tasks scheduler receives a heartbeat from a node that has an idle slot, a task is scheduled to it instantly. If a job's data is only stored on a small fraction of nodes, during job execution, a task is scheduled to the node that sends heartbeat first rather than the node with data. Therefore task locality can be very poor. For example, if a job has data on 20% of nodes, only 20% tasks are local tasks on average.

The second problem is that in rack-local and non-local task sets, naive scheduling just simply picks the first available task without further considering current cluster status, especially network conditions. The execution time of some tasks may be longer than other similar tasks because of network congestion and these outlier tasks will influence the performance of the whole system.

B. Delay Scheduling

Delay scheduling [17] was proposed to solve the first problem in naive scheduling. The main idea of delay scheduling is to delay launching a non-local task. During the delay, task scheduler may find a data-local node to execute the current task. Algorithm 2 shows the pseudo code of delay task scheduling being used in Apache Spark.

Using a simple technique, delay scheduling solves the locality problem of naive scheduling in small jobs. But in large jobs, if the data distribution is unbalanced, the second problem of naive scheduling still exists. For example, in figure 2 if a job has no data partition on node A and twenty partitions in other nodes, and each node has 10 slots to execute tasks. We

Algorithm 2: Delay task Scheduling

```
Input: Node $n$ that has an idle slot, rack-local waiting
       time $T_r$, non-local waiting time $T_{na}$
Output: Task $t$ to be assigned to $n$
Initialize j.lastlaunchtime to current time for all jobs j.
for $j$ in jobs do
    if $j$ has unassigned task $t$ with data on $n$ then
        set j.lastlaunchtime to current time ;
        return t;
    end
    else if $j$ has unassigned task $t$ with data on nodes in
    the same rack with $n$ then
        if $currenttime - j.lastlaunchtime > T_r$ then
            return t;
        end
    end
    else if $j$ has unassigned task $t$ then
        if $currenttime - j.lastlaunchtime > T_{na}$ then
            return t;
        end
    end
end
return null;
```

assume the size of each partition is 1Gb. Here we use a concept called data processing rate (hereinafter referred to as DPR) to measure the computation intensity of a task. DPR means the amount of data that can be processed per second by the current CPU if the data I/O is unlimited. This metric is influenced not only by task property but also by CPU's capability. So we usually say DPR under current environment. We assume the DPR of the tasks under current environment is 1Gb/s. If node A is not in the cluster, all tasks are executed locally and it only takes 4 seconds to complete them. But when we add node A to the cluster, tasks will be scheduled to node A after waiting a small amount of time. Node A will have 10 non-local tasks to execute. If the incoming network bandwidth of node A is only 1Gb/s, the tasks will congest on node A which means their actual executing rate is only 100Mb/s. It will take 10 seconds to finish them in this case. From this scenario, we can see that unbalanced data distribution will hurt the performance of the cluster. It takes longer to finish the tasks when we add a new node to the cluster which makes data unbalanced. This task scheduling is worse than ignoring node A and executing all tasks locally. And it can be even worse if there is neighborhood traffic from other nodes to node A.

In delay-based scheduling, one method to solve this problem is to increase waiting time to achieve better locality. Figure 3 shows an example of the relationship between waiting time and locality. According to 2, if the waiting time is larger than task execution time, no non-local task will be scheduled and all tasks will be local tasks. However, although longer waiting time increases data locality, it also means waste of computing resource. So in practice, we need to balance these two factors and find the optimal waiting time to obtain the best performance. Since for different jobs (data intensive, CPU intensive) and different data locality (well balanced, unbalanced) the optimal waiting time is different, it's impossible to find an best value that fits all scenarios. It's also hard to find the best waiting time each time before a job starts because this requires knowledge of the CPU DPR before a job starts but CPU DPR varies even in different stages of the same job.

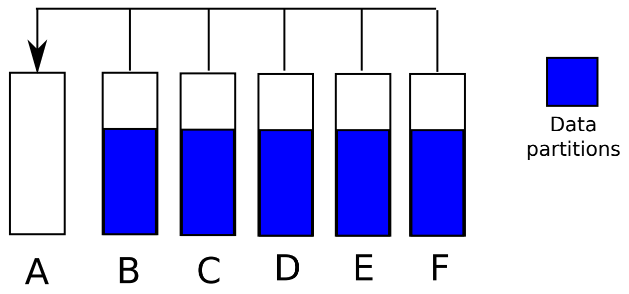


Fig. 2. Examples of unbalanced data distribution

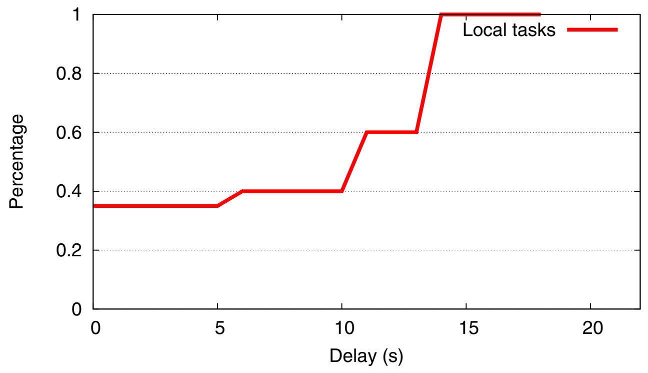


Fig. 3. Relationship between waiting time and locality

C. Network-aware Scheduling

The problems occur since existing scheduling methods don't consider network status in a cluster. We observed that network congestion mainly occurs at nodes with fewer data blocks or with large incoming neighborhood traffic which results in network congestion at the node. To solve this problem, we propose a new network-aware scheduling method which aims to avoid network congestions and achieve efficient scheduling to increase the throughput of the whole system. Our network-aware task scheduler is implemented based on delay scheduler and uses the global network status to help make scheduling decisions. When a node requires a task, if there is no local task to schedule, the scheduler will first check if the node has enough available bandwidth to accommodate a new non-local task. If so, the scheduler finds a task with enough available bandwidth from data node to the executor node, otherwise, no task is scheduled at this time. The pseudo code of our method is shown in Algorithm 3.

Compared with current delay scheduling method, our method has three advantages. The first is that while scheduling non-local tasks, it considers the network overhead of the node that has idle slots, which can prevent scheduling too many non-local tasks to the same node. The second advantage is that since the scheduler can get global network status, it can pick a non-local task that will not experience congestion during data shipping, which accelerates the execution process. Finally, our method can achieve better performance than existing scheduling methods without any system tuning or parameter adjustment when executing different kind of jobs.

Algorithm 3: Network-aware task Scheduling

```
Input: Node $n$ that has an idle slot, rack-local waiting
       time $T_r$, non-local waiting time $T_{na}$
Output: Task $t$ to be assigned to $n$
Initialize j.lastlaunchtime to current time for all jobs j.
for $j$ in jobs do
    if $j$ has unassigned task t with data on $n$ then
        set j.lastlaunchtime to current time ;
        return t;
    end
    else if $currenttime - j.lastlaunchtime > T_r$ then
        for task t in j.rack-local do
            compute t.DPR
            compute availableCapacity which equals
            bandwidth of the best path from t.split to n
            if availableCapacity>t.DPR then
                return t;
            end
        end
    end
    else if $currenttime - j.lastlaunchtime > T_{na}$ then
        for task t in j do
            compute t.DPR
            compute availableCapacity which equals
            bandwidth of the best path from t.split to n
            if availableCapacity>t.DPR then
                return t;
            end
        end
    end
end
return null;
```

D. Analysis of scheduling methods

The previous sections provide an overview of different scheduling method. In this section, we analyse and compare these methods in detail. We first summarize these schedulers as three classes of scheduling models and then analyse their performance in theory.

1) Scheduling Models: In Spark, scheduling methods can be separated into three classes based on how they deal with unbalanced data locality.

Non-local preferred scheduling The key idea of this model is that when there is no local task on the idle executor, it prefers to schedule a non-local task. Both naive scheduling and delay scheduling with small waiting time can be seen as non-local preferred scheduling. If network capacity is adequate and there is no data contention, then there is no overhead caused by unbalanced locality because non-local tasks can be executed at the same processing rate as local tasks. If data processing rate is too large or network capacity is inadequate, there will be overhead because network capacity is not sufficient to accommodate non-local tasks and data contention will cause delay of non-local tasks.

Local preferred scheduling In this model, when there is no local task on the idle executor, it prefers waiting to scheduling a non-local task. Thus, most tasks are executed locally and non-local tasks are avoided during processing. Delay scheduling with long waiting time can be seen as local preferred scheduling. Since extra data on some nodes can not be digested by other nodes and nodes with fewer data chunks are not well utilized, the overhead will be high if data locality is highly unbalanced.

Adaptive scheduling The key idea of adaptive scheduling is that while scheduling non-local tasks, scheduler considers network capacity and DPR of current tasks to avoid data contention caused by either unbalanced locality or limited network capacity. Our network-aware scheduling can be seen as adaptive scheduling.

2) Scheduling Analysis: In this section, we analyze different scheduling models for unbalanced-locality datasets. Table I lists the notations for this subsection.

M	Number of servers
D	Data size (bytes)
R	DPR of all cores of a server (bytes/s)
C	Network bandwidth of a server (bytes/s)
α	Percentage of local tasks

TABLE I. NOTATIONS

In Spark, the execution time of a job is determined by the last finished task. Unbalanced locality means that some nodes have more data chunks while others have less. If one node has much more data chunks than the other nodes, then it has to ship its data to other nodes so that its outgoing network can be a bottleneck. If one node has much less data chunks than the other nodes, then it has to get data from other nodes so that its incoming network can be a bottleneck. Thus we analyze how scheduling methods perform and how network-aware scheduler achieves optimal scheduling in these two scenarios.

We first analyze an extreme version of the first scenario: one node has all data and there is no data on other nodes. These tasks can either be scheduled as local tasks or non-local tasks. We assume α is the percentage of local tasks and $(1-\alpha)$ is the percentage of non-local tasks. The execution time of tasks on Node 1 is $\frac{\alpha D}{R}$ and the execution time of tasks on other nodes is $\frac{(1-\alpha)D}{min(C,(M-1)R)}$. So the execution time of all tasks should be

$$T = max(\frac{\alpha D}{R}, \frac{(1-\alpha)D}{min(C,(M-1)R)}) \tag{1}$$

According to this equation, we can find that the only difference among different scheduling methods is α and we'll see how α is determined in different scheduling models.

Under non-local preferred scheduling, α is generally $\frac{1}{M}$ which means tasks are evenly distributed to all nodes. And

	$(M-1)R < C$	$(M-1)R \geq C$
Non-local Preferred	$\frac{D}{MR}$	$\frac{(M-1)D}{MC}$
Local Preferred	$\frac{D}{R}$	$\frac{D}{R}$
Network-aware	$\frac{D}{MR}$	$\frac{D}{R+C}$

TABLE II. EXECUTION TIME OF THREE SCHEDULING MODELS IN SCENARIO 1

the execution time is:

$$T_{np} = \begin{cases} \frac{D}{MR} & \text{if } (M-1)R < C, \\ \frac{(M-1)D}{MC} & \text{if } (M-1)R \geq C \end{cases}$$

Under local preferred scheduling, α is generally a value close to 1, so we just use 1 here. This means that no non-local task is assigned and all tasks are executed locally. The execution time is:

$$T_{lp} = \frac{D}{R}$$

Under network-aware scheduling, α is determined by data processing rate and network capacity. If $(M-1)R < C$ which means network capacity is sufficient for all non-local tasks, $\alpha = \frac{1}{M}$. If $(M-1)R \geq C$ which means network capacity is not enough, we only assign non-local tasks that can be digested by network. Since local task DPR is R and non-local task DPR is C, the fraction of local tasks is $\alpha = \frac{R}{R+C}$. So the execution time is:

$$T_{na} = \begin{cases} \frac{D}{MR} & \text{if } (M-1)R < C, \\ \frac{D}{R+C} & \text{if } (M-1)R \geq C \end{cases}$$

Table II gives a comparison of the three models. From this table, we can find that while $(M-1)R < C$, non-local preferred and network-aware methods performs M times better than local preferred method because they make use of those nodes without data to help accelerate execution. But while $(M-1)R \geq C$, there is difference between non-local preferred method and network-aware method. If we assume M is very large, the execution time of non-local preferred method is approximately $\frac{D}{C}$. For some data-intensive jobs like TPC-H queries, $C << R$, so non-local preferred method can be much worse than local preferred method because of data contention. Network-aware method can double outperform both non-local preferred method and local preferred method under the condition $C \approx R$.

Next in the second scenario, most nodes have the same amount of data and there is no data on one node (Fig 2). We assume that α is the percentage of local tasks and all non-local tasks are executed by Node 1. The execution time of tasks on Node 1 is $\frac{(1-\alpha)D}{min(R,C)}$ and the execution time of tasks on other nodes is $\frac{\alpha D}{(M-1)R}$. So the execution time of all tasks should be

$$T = max(\frac{(1-\alpha)D}{min(R,C)}, \frac{\alpha D}{(M-1)R}) \quad (2)$$

We will also analyze different methods by finding how α is determined.

Under non-local preferred scheduling, α is generally $\frac{M-1}{M}$ which means tasks are evenly distributed to all nodes. And the execution time is:

$$T_{np} = \begin{cases} \frac{D}{MR} & \text{if } R < C, \\ \frac{D}{MC} & \text{if } R \geq C \end{cases}$$

Under local preferred scheduling, α is generally a bigger value, so we just use 1 here. This means that no non-local task is assigned and all tasks are executed locally. The execution time is:

$$T_{lp} = \frac{D}{(M-1)R}$$

Under network-aware scheduling, if $R < C$ which means network capacity is sufficient for all non-local tasks, $\alpha = \frac{M-1}{M}$. If $R \geq C$ which means network capacity is not enough, we only assign non-local tasks that can be digested by network. The fraction of local tasks is $\alpha = \frac{(M-1)P}{(M-1)P+C}$. So the execution time is:

$$T_{na} = \begin{cases} \frac{D}{MR} & \text{if } R < C, \\ \frac{D}{R+C} & \text{if } R \geq C \end{cases}$$

	$(M-1)R < C$	$(M-1)R \geq C$
Non-local Preferred	$\frac{D}{MR}$	$\frac{D}{MC}$
Local Preferred	$\frac{D}{(M-1)R}$	$\frac{D}{(M-1)R}$
Network-aware	$\frac{D}{MR}$	$\frac{D}{(M-1)R+C}$

TABLE III. EXECUTION TIME OF THREE SCHEDULING MODELS IN SCENARIO 2

Table III gives a comparison of the three models in this scenario. From this table, we can find that while $(M-1)R < C$, the 3 methods performs almost the same when M is large. But while $R \geq C$, especially if $R >> C$, the execution time of non-local preferred method is much worse than the other 2 methods.

From our above analysis, we can see that non-local preferred method is weak in data-intensive jobs and local preferred method is weak in CPU-intensive jobs. But our network-aware method performs well in both cases and achieves optimal scheduling. Here we only picks 2 simple scenarios to show a theoretical comparison. In practice, there can be more complicated circumstances and the randomness of task assignment may also affect the performance. We do not claim network-aware scheduling can always achieve optimal scheduling. And usually in real data centers, there is data on every node, so all nodes would execute local tasks at the beginning, and non-local tasks emerge when there is no more local task on an idle node. So the difference between the 3 scheduling methods may be observed in the latter portion of a stage's execution.

IV. IMPLEMENTATION

In this section, we first introduce the overall architecture of our implementation and then describe the functionality of each component in detail. And finally we show how they work together to achieve network-aware scheduling in a real Spark cluster.

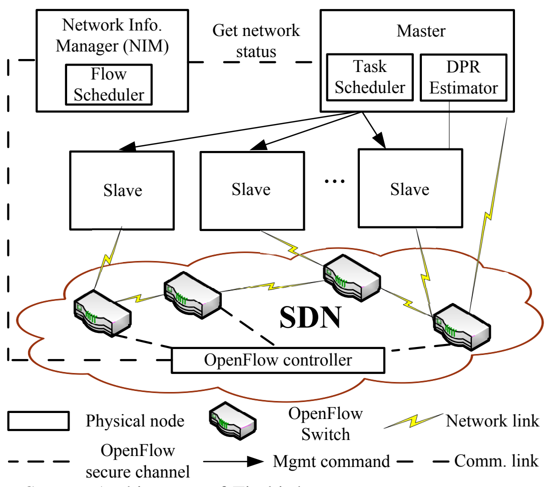


Fig. 4. System Architecture of Firebird

A. Architecture

We implemented a system called Firebird using Spark 1.3.1 in conjunction with software-defined network. Our system consists of two modules: network module and scheduling module. The network module is a network information manager (NIM) and flow scheduler. The scheduling module consists of two components: DPR estimator and task scheduler. Figure 4 shows the architecture of our system.

B. Network Information Manager

The NIM enquires and updates information and state of the current network by communicating with the OpenFlow controller. The network information includes the network topology (hosts, switches, ports), queues, links, and their capabilities. The NIM also hosts the switch information such as its ports' speeds, configurations, and statistics. It is important to keep this information up-to-date as inconsistency could lead to under-utilization of network resources as well as poor task scheduling. The NIM maintains a network status snapshot by collecting traffic information from OpenFlow switches periodically.

When a scheduler sends an inquiry to the NIM to inquire the available bandwidth between M and N, the NIM returns the maximum bandwidth of all paths from M to N.

C. Flow scheduler

In NIM, we follow the flow scheduler design in Hedera [1], i.e., the scheduler aims to assign flows to *nonconflicting* paths.When a new flow arrives at one of the edge switches, according to OpenFlow protocol, a "PacketIn" message is sent from the switch to the controller. The first packet of the flow is delivered to the controller. The controller examines the 10 tuple packet information and forwards the information to the flow scheduler. Based on the packet information, the flow scheduler can identify the source and the destination of the flow. The flow scheduler will compare the available bandwidth for all the candidate physical paths for this flow. It then chooses the best path that has the maximum available bandwidth (i.e., the least conflicting paths). The best path is decomposed with multiple hops. For each hop, the flow scheduler uses the path change handler to configure the path, i.e., asks the OpenFlow controller to send "FlowMod" message to the switch to modify a switch flow table. Note that, we again assume that more available bandwidth will make the flow run faster and this approach is only "local" optimal for this flow but may not be "global" optimal for all the flows.

In our initial experiments, we find that in a dedicated network, the network bottleneck is usually the node network capacity rather than intermediate path capacity. So flow scheduler shows little improvement. But when network is shared with other applications, the flow scheduler can significantly reduce the risk of network contention. So we implement flow scheduler to enable our system to fully utilize network resource in both cases.

D. DPR Estimator

In traditional cluster computing platforms like Hadoop, since the bottleneck for data intensive jobs is disk I/O, it's hard to estimate CPU data processing rate. Since network capacity is not a problem, it's also unnecessary to schedule non-local tasks according to DPR. But in in-memory cluster computing platforms such as Spark, things are different. Memory I/O replaces disk I/O which means local I/O is much faster and is no longer the bottleneck. Hence we can compute CPU DPR of a finished task by simply dividing data size by execution time. For simplicity, we assume that all nodes in the cluster are homogeneous. We observe that the DPRs of tasks in the same stage of a job are similar. So the DPR of a new task can be estimated as the average DPR of finished tasks in current stage.

E. Task Scheduler

For process-local and node-local tasks, since they don't generate network traffic, our task scheduler follows the design of default task scheduler. According to Algorithm 3, when a node is idle, the scheduler first searches for tasks in node-local sets. If there is no task returned, it continues to search in rack-local and non-local task set. For each task in rack-local and non-local task sets, DPR estimator first estimates the bandwidth that the task requires. Then task scheduler inquires NIM to get the available bandwidth between data node and execution node. Since the real flows are generated several seconds after tasks are scheduled, to avoid contention of simultaneously scheduled tasks, task scheduler also records the reserved bandwidth for scheduled tasks before the real flows are generated. By combining NIM's bandwidth information and reserved bandwidth, task scheduler computes the real available bandwidth to see if it is enough to accommodate this task. If so, the task is scheduled to the idle executor, and if not, it tries the next task. If there are no tasks that can be scheduled, the executor remains idle until next heartbeat.

V. Experimental Evaluation

In this section, we describe our experimental settings and the experimental results.

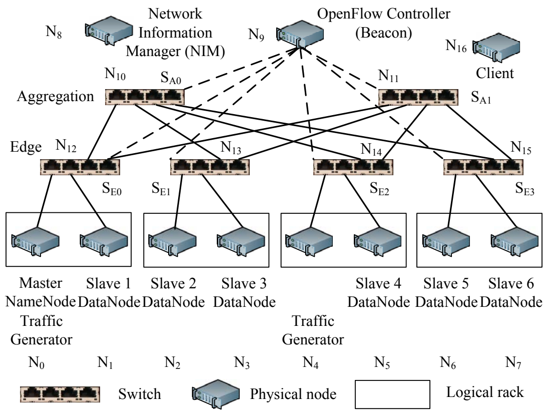


Fig. 5. Spark Cluster Setup

A. Experimental Settings

1) Hardware settings: Our experimental testbed as shown in Figure 5 consists of 17 physical nodes N_{0-16}. Each of the machines has an Intel Xeon E5-2440 2.4GHz six core CPU, 32GB of RAM, 1TB 7200rpm disk running Linux with kernel 2.6.32. Each of the machines has two Gigabit ethernet NICs. Six of the machines N_{10-15} are installed with a 4-port Gigabit NetFPGA card serving as a 4-port OpenFlow switch. Hence, N_{10-15} operate as OpenFlow switches. Seven of the machines N_{1-3}, N_{5-7} are used for both Spark and HDFS deployment with one master (at N_0) and six slaves (at N_{1-3}, N_{5-7}). N_4 is used for generating neighborhood network traffic. N_8, N_9 and N_{16} are used to run network information manager(NIM), Openflow Controller, and client emulator, respectively.

There are two networks in our testbed, a management network and an OpenFlow network. The first NIC of each machine is connected by a Gigabit Cisco switch, which forms the management network. The second NIC of N_{0-7} is connected by a 4-port Gigabit NetFPGA OpenFlow switch, which forms the OpenFlow network. All the Spark traffic goes through the OpenFlow network. We use an open source OpenFlow controller Beacon [1] as our OpenFlow controller. The OpenFlow network is built following a similar network topology in prior work [1] to enable multiple paths from a source host to a destination host.

2) Benchmark: We use 3 different types of jobs: TPC-H Q6, Kmeans and Word Count as examples to evaluate the performance of our method.

TPC-H Query 6 TPC-H queries are known as data intensive jobs and the DPR of core is 500Mb/s-800Mb/s in our experimental environment. We pick Query #6 (Q6) because it's easy to analyze: its reduce phase is very short and most of execution time is spent on map tasks. We show details of Q6 below.

```
select
  sum(l_extendedprice*l_discount) as revenue
from
  lineitem
where
  l_shipdate >= '1994-01-01'
  and l_shipdate < '1995-01-01'
  and l_discount >= 0.05 and l_quantity < 24
  and l_discount <= 0.07;
```

[1]https://openflow.stanford.edu/display/Beacon/Home

We use TPC-H dataset with scaling factor of 100GB generated by dbgen and the chunk size is 512MB. This query is a single-stage job with a lot of map tasks but only 1 reduce task.

Word Count Word count is a less data intensive job compared with TPC-H queries. The DPR of one core is 250Mb/s-350Mb/s in our experimental environment. In our experiment, the dataset of word count is 50G and the chunk size is 512MB. It's also a single-stage job.

Kmeans In contrast to TPC-H and Word Count, Kmeans is a multi-stage CPU intensive job. Since it's multi-stage, the DPR varies in different stages, but in the range of 10Mb/s-200Mb/s in our experimental environment. In our experiment, the dataset size is 5GB and consist of 30 clusters of 3D points. The chunk size of the dataset is 64MB.

We run each job multiple times and discard the result of the first run because in the first run data is not cached in memory.

B. Dedicated Network

In this section, we first consider an idle (dedicated) network that only sees Spark traffic. We compare the performance of three types of schedulers mentioned in section III-D. Default delay scheduler with a delay of 3 seconds performs as non-local preferred scheduler. Local preferred scheduler is implemented by setting the waiting time longer than task execution time so that non-local tasks are not scheduled. In order to test Firebird's performance under different environment, we vary the number of cores that can be used on each server by varying the configuration file of Spark.

Table IV shows the number of data chunks on each node for each workflow. We can see from the table that naturally distributed data is not well balanced.

	S1	S2	S3	S4	S5	S6
TPC-H Q6	28	21	17	24	31	28
Word Count	22	19	12	16	19	18
K-Means	15	11	10	19	12	16

TABLE IV. NUMBER OF CHUNKS ON EACH NODE.

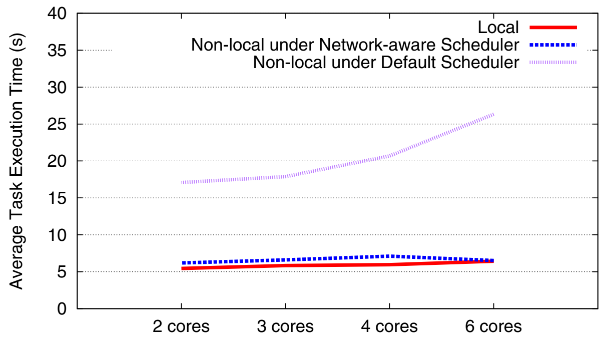
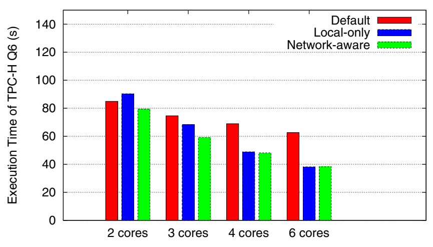
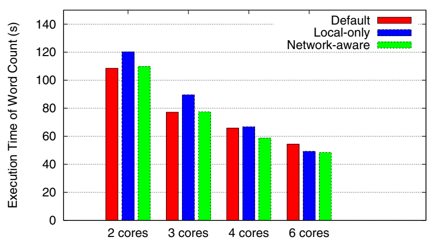


Fig. 7. Average execution time of local and non-local tasks

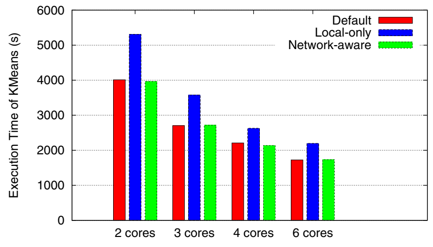
Figure 6 shows the performance of the three schedulers with different number of cores. From figure 6(a), we can see that network-aware scheduler performs almost the same as



(a) Execution time of TPC-H q6



(b) Exectution time of word count



(c) Execution time of Kmeans

Fig. 6. The execution time of different jobs while using 3 different schedulers

local preferred scheduler and up to 39% better than non-local preferred scheduler in 4 core and 6 core scenarios. Network-aware scheduler performs 5-20% better than the other two schedulers in 2 core and 3 core scenarios.

From figure 6(b), we can see that our network-aware scheduler is 10% better than local preferred scheduler in 4-core and 6-core scenarios and is 10% better than non-local preferred scheduler in 2-core, 3-core and 4 core scenarios.

From figure 6(c), we can see that for CPU intensive jobs like KMeans, local preferred scheduler performs worse than the other 2 schedulers. In 2-core scenario, local preferred scheduler is 24% worse than the other 2 schedulers. This result justifies our conclusion in section III-D2.

The experimental results are in line with our analysis in section III-D2 that non-local preferred method is weak in data-intensive jobs and local preferred method is weak in CPU-intensive jobs. From these 3 graphs, we can also see that our network-aware scheduler is always the best of the three schedulers no matter the job is data intensive or CPU intensive. When the job is CPU intensive and the number of cores is small, network-aware scheduler performs similar to non-local preferred scheduler. W the job is data intensive and the number of cores is larger, network-aware scheduler performs similar to local preferred scheduler. Also network scheduler performs better than both for scenarios in between.

Figure 7 shows the average execution time of tasks in TPC-H Q6. We can see that as the number of cores increases, when non-local preferred scheduler is used, the execution time of non-local tasks increases because of network contention. While using network-aware scheduler, the execution time of non-local tasks doesn't increase and is still close to local tasks. This means that scheduling tasks according to available bandwidth is reasonable and it can avoid outlier tasks efficiently.

These experiments show that our network-aware scheduler is efficient for both CPU intensive and data intensive jobs. Hence it doesn't need any system tuning while executing different jobs which is very convenient to users in practice.

C. Shared Network

In real data centers, network is often shared by different types of applications. The performance of Spark can be influenced by network traffic generated by other applications. We expect that by being aware of current network status, our network-aware scheduler can improve Spark's performance under shared network.

To create background traffic, we add a flow generated by iperf from an idle node (traffic generator) to N3. From figure 8, we can see that as the iperf flow becomes larger, the execution time of using default scheduler is highly influenced. Since N3 has less data than other nodes, after local tasks are finished, it would be assigned non-local tasks and read data from other nodes. Since the background traffic limits the available bandwidth that N3 can use, the data transfer speed of these non-local tasks is slow and the execution time of the whole job is influenced. While the iperf flow is 800Mb/s, the execution time becomes 5 times longer in TPCH Q6 and 3 time longer in Word Count than that with no background traffic. But network-aware scheduler is not influenced much by this background traffic because it can acquire the network status and avoid assigning non-local tasks to N3. The experiment shows that network-aware scheduler performs 9 times better in TPCH Q6 and 3 times better in Word Count than default scheduler under 800Mb/s background traffic.

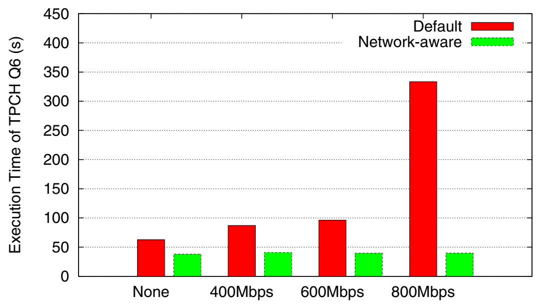
VI. Related Work

In this section, we discuss related work from two areas: scheduling and networking.

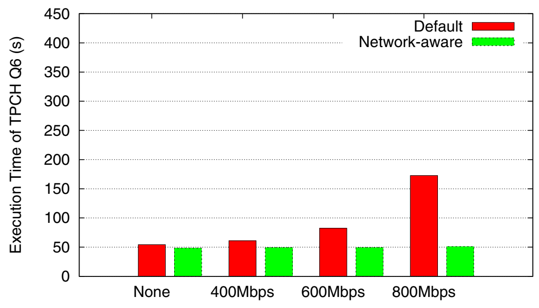
In the scheduling context, there has been a plethora of work on optimizing MapReduce performance. CoHadoop [9] extends HDFS and adds metadata to NameNode so that it allows applications to control where data are stored. HaLoop [4] caches the reused data on local disks and then improves performance by leveraging this data locality in the scheduler. Hyracks [3] is a data-parallel runtime platform designed to perform data-processing tasks on large amounts of data using large clusters. Because data is processed in a pipelined manner, Hyracks can push a very large amount of data to the network thereby potentially creating extreme network contention during the run-time.

All of the above work treats the network as a black box. Our contribution is complementary by considering network status when scheduling task.

From the networking perspective, Wang *et al.* [15] propose application-aware networking, and argue that distributed applications can benefit from communicating their preferences to the network control-plane. Yap *et al.* have also advocated for an explicit communication channel between applications



(a) Execution time of TPCH Q6 under outside network traffic



(b) Execution time of WordCount under outside network traffic

Fig. 8. Network traffic generated by other applications can largely influence the performance of Apache Spark

and software-defined networks, in what they called software-friendly networks [16]. PANE [10] proposes design, implementation, and evaluation of an API for applications to control a software-defined network. Sinbad [5] is a system that identifies imbalance and adapts replica destinations to navigate around congested links. It can be seen as a network-aware data placement method. Coflows [6] is a a networking abstraction that allows cluster applications to convey their communication semantics to the network. Based on [6], Varys [7] enables data-intensive frameworks to use coflows to optimize network scheduling. Both [5] and [7] assume data center is dedicated and all network flows can be controlled by their system. While our work focuses on how to use network-aware task scheduling to improve Spark's performance in both dedicated and shared network environments with dynamic uncontrolled flows.

VII. Conclusion and Future Work

In this paper, we propose a network-aware scheduling method that exploits software-defined networking and implement it in Spark. Unlike previous work, in which Spark works independently from the network underneath, our system enables Spark and the networking layer to work together to improve network utilization and reduce job execution times. As our experiments show, this improvement can be huge when network contention is heavy. We show that our network-aware scheduling method doesn't require any system tuning when facing different kind of jobs.

We see many avenues for future work. For example, currently we only focus on task scheduling of a single job. It could be interesting to find how to optimally schedule tasks of different jobs with different locality properties. Furthermore, if the jobs have different priority, the problem becomes more complicated and new policies need to be designed.

References

[1] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *Proc. of NSDI*, 2010.

[2] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *Proc. of OSDI*, 2010.

[3] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *Proc. of ICDE*, 2011.

[4] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. Haloop: Efficient iterative data processing on large clusters. In *Proc. of VLDB*, 2010.

[5] M. Chowdhury, S. Kandula, and I. Stoica. Leveraging endpoint flexibility in data-intensive clusters. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 231–242, New York, NY, USA, 2013. ACM.

[6] M. Chowdhury and I. Stoica. Coflow: a networking abstraction for cluster applications. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pages 31–36. ACM, 2012.

[7] M. Chowdhury, Y. Zhong, and I. Stoica. Efficient coflow scheduling with varys. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 443–454, New York, NY, USA, 2014. ACM.

[8] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proc. of OSDI*, 2004.

[9] M. Y. Eltabakh, Y. Tian, F. Özcan, R. Gemulla, A. Krettek, and J. McPherson. Cohadoop: Flexible data placement and its exploitation in hadoop. In *Proc. of VLDB*, 2011.

[10] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi. Participatory networking: An api for application control of sdns. In *Proc. of SIGCOMM*, 2013.

[11] B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, and L. Zhou. Comet: Batched stream processing for data intensive distributed computing. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 63–74, New York, NY, USA, 2010. ACM.

[12] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 2008.

[13] Open Networking Foundation. Software-Defined Networking: The New Norm for Networks. 2013.

[14] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha. Sharing the data center network. In *Proc. of NSDI*, 2011.

[15] G. Wang, T. E. Ng, and A. Shaikh. Programming your network at run-time for big data applications. In *Proc. of HotSDN*, 2012.

[16] K.-K. Yap, T.-Y. Huang, B. Dodson, M. S. Lam, and N. McKeown. Towards software-friendly networks. In *Proc. of APSys*, 2010.

[17] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*, pages 265–278. ACM, 2010.

[18] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.