Cormorant: Running Analytic Queries on MapReduce with Collaborative Software-defined Networking

Pengcheng Xiong*, Xin He†, Hakan Hacigumus‡, Prashant Shenoy†
*Hortonworks, Inc
{pengcheng.xiong}@gmail.com
†University of Massachusetts Amherst
{xhe, shenoy}@cs.umass.edu
‡Google, Inc
{hakanh}@gmail.com

***Abstract*—MapReduce is a popular choice for executing analytic workloads over large datasets on clusters of commodity machines. Due to the distributed nature of such systems, network resource bottlenecks can adversely affect performance, especially when multiple applications share the network. One of the significant barriers to reducing the occurrence and impact of such bottlenecks is the current separation between MapReduce and network management and routing. Fortunately, the emergence of software-defined networking (SDN) is removing the barriers to cooperation between Hadoop and the network. To explore the opportunity this creates, we focus on how we can use the capabilities of SDN to create a more collaborative relationship between MapReduce and the network underneath. We demonstrate the effectiveness of this collaboration through the implementation of and experiments with a system we call Cormorant. Experimental results show up to 38% improvement for analytic query performance, beyond the benefits achievable by independently optimizing MapReduce schedulers and network flow schedulers.**

I. Introduction

Running analytic queries on large, diverse, and ever-growing datasets, so-called big data processing, has become an essential part of business processes for enterprises. MapReduce [4] (and Hadoop as the open source version of MapReduce) 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. However, due to the distributed nature of the framework, the network bandwidth resource has always been a scarce resource that limits the MapReduce's performance [2]. Moreover, this problem becomes even more challenging if the network is shared with other applications as well [8].

One cause of the problem is the current separation between the decisions MapReduce and networking make with respect to resource allocation. MapReduce does not explicitly monitor the underlying network status, nor does it try to modify its activities due to this status. Similarly, the networking layer does not base its resource allocation based on any insight into the specific expected behavior of a MapReduce task. As a result, when the network is shared with other applications, it simply tries to deliver its network service to all applications equally. In this paper we explore the issue of whether or not higher performance can be obtained by changing the fundamental relationship between MapReduce and network routing by exploiting the cooperative capabilities offered by software-defined networking (SDN) [6], [7]. We focus on MapReduce workloads generated by Hive as representative of a widely used approach to executing decision support queries over large data sets.

Data center applications initiate connections between a diverse range of hosts and can require significant aggregate bandwidth. Data center topologies often implement a multi-rooted tree with higher-speed links but decreasing aggregate bandwidth moving up the hierarchy[1]. These multi-rooted trees have many paths between all pairs of hosts. A key challenge is to simultaneously and dynamically forward flows along these paths to minimize or reduce link oversubscription and to deliver acceptable aggregate bandwidth. Unfortunately, existing network forwarding protocols are optimized to select a single path for each source/destination pair in the absence of failures. Such static single-path forwarding can significantly underutilize multi-rooted trees. The state of the art forwarding in data center environments uses ECMP [5] (Equal Cost Multipath) to statically stripe flows across available paths using flow hashing. This static mapping of flows to paths does not account for either current network utilization or individual flow size.

Recently, Hedera [2] has been proposed as a dynamic flow scheduling system for generic workloads in data centers with multi-rooted tree topologies. Hedera is a substantial improvement over the network status and flow size oblivious

This work was done at NEC Labs America.

[1]Cisco Data Center Infrastructure 2.5 Design Guide. www.cisco.com/univercd/cc/td/doc/solution/dcidg21.pdf.

ECMP algorithm. In view of this, we have chosen Hedera as our flow scheduler and use it in our experiments.

However, Hedera only goes part of the way to collaborative, network-aware scheduling between task managers and flow schedulers. That is, Hedera is invoked after the tasks have already been selected, and seeks to schedule and route the resulting flows given that the tasks selected are fixed. In this work we aim to discover if completing the transition is effective – that is, if an even tighter integration between the task manager and the flow scheduler can yield better performance.

Another important general idea in reducing the impact of bandwidth limitations in Map Reduce computations is to place jobs "close to" their data, thus reducing the amount of data that must be transferred. A relevant piece of work along these lines is Mantri [3]. Mantri can yield much better performance than location-oblivious placement of tasks; in view of this, we have implemented Mantri as our task scheduler. However, while Mantri impacts the flows that a particular application will generate, it does not manage those flows, nor does it monitor or react to the status of the network. As such, Mantri is also complementary to our work.

Thus, the main goal of this paper is to explore the following: given that we are using a state-of-the-art flow scheduling algorithm (Hedera) and a state-of-the-art task-placement algorithm (Mantri), is there still room for further improvement by exploiting the capabilities of software-defined networking (SDN) to establish a collaborative relationship between a system executing decision support queries over Hadoop and the network providing the communication below? We provide an initial answer of "yes" and also lay the groundwork for future follow-on work exploring this question.

Leveraging SDN for better performance of analytical queries was also considered in [9]. However, the scenarios considered in [9] are limited to traditional relational query processing, while our work focuses on MapReduce systems. One important difference is that in relational systems, query processing and the storage management are tightly coupled, whereas in Hadoop-based systems they are managed separately (MapReduce processing and HDFS file system). This separation calls for different management and optimization techniques for task and flow scheduling, which we study in this work.

Our contributions: In this paper, we propose Cormorant, a Hadoop-based query processing system built on top of collaborative software-defined networking. Different from the previous work, our work aims at building a collaborative relationship between MapReduce and networking. MapReduce optimizes task schedules based on the information provided by software-defined networking and software-defined networking guarantees the exact schedule to be executed. To the best of our knowledge, this is the first paper to analyze and show the power and opportunities of collaborative software-defined networking for a MapReduce system. It is our hope that this will open up a rich area of research and technology development in distributed data intensive computing.

Our specific contributions over the existing work are the following:

(1) We present a method that enables task scheduler in Hadoop to obtain current network status to improve the task scheduling.

(2) We present a flow scheduler to dynamically change the physical flow path following Hadoop's requirements. Moreover, we make the task scheduler work with the flow scheduler collaboratively.

(3) We have implemented all of the techniques in Hadoop running on a "real" software-defined networking with OpenFlow enabled switches. Our experimental results show that Cormorant can effectively reduce the overhead incurred by network congestion and achieve 14-38% improvement for TPC-H query execution. The benefits of collaboration go beyond simply adding up the benefits of a work-alone task scheduler and a work-alone flow scheduler from prior work.

II. Background

In this section, we give a brief background on some components to help presentation of the methods in the sequel.

A. Apache Hadoop

We use the standard Apache Hadoop distribution as the basis for the implementation of the system. We use HiveQL as the high level query language to express the database queries. A HiveQL query is translated into MapReduce jobs to run on Hadoop. For each MapReduce job, the Hadoop Distributed File System (HDFS) handles the reading/writing of the job's input/output data. A MapReduce job is generally composed of several map tasks (mappers) and reduce tasks (reducers). For each mapper, it reads a chunk of input data from HDFS, applies the map function to extract key, value pairs, then shuffles these data items to partitions that correspond to different reducers, and finally sorts the data items in each partition by the key. For each reducer, when it has collected all of the map output, it will perform a merge to produce all key, value pairs in sorted order of the key. As the merge proceeds, the reducer applies the reduce function to each group of values that share the same key, and writes the reduce output back to HDFS.

B. Multi-rooted tree network topologies

Today's data centers could consist of thousands of connected servers. The recent research advocates multi-rooted tree topologies [1], where there are a larger number of parallel paths between any given source and destination edge switches. One rational for the existence of multiple paths is to achieve fault tolerance.

For example, Figure 1 shows three layers of switches, i.e., edge layer, aggregation layer, and the core layer. The edge layer switches directly connect to the servers. We can see that there are a larger number of parallel paths between any given source and destination edge switches. Note that, although 10Gb links are used in the aggregation layer and the core layer, which are more powerful than 1Gb links in the edge layer, it

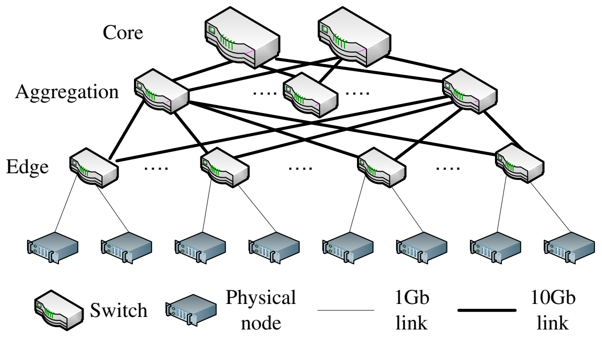

Fig. 1. A common multi-rooted hierarchical tree

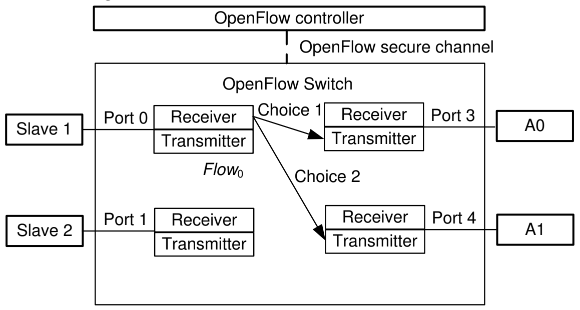

Fig. 2. Inside an OpenFlow switch

is currently very hard to achieve full bisection bandwidth due to the high oversubscription factor [1].

C. Software-defined networking and OpenFlow

SDN is an 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 [7]. From the Hadoop 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 communication interface among the layers of an SDN architecture, which can be thought of an enabler for SDN [6]. 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.

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.

For example, we show a 4-port OpenFlow switch S_{E0} serving as an edge switch in Figure 2. Two nodes $N_{0,1}$ are connected to S_{E0} at ports 0,1 and two aggregation switches S_{A0} and S_{A1} are connected to the switch at ports 2,3, respectively. There is a receiver and a transmitter behind each port of the switch. When a new flow $Flow_0$(from N_0 to N_2) arrives, a "PacketIn" message is sent from the switch S_{E0} to the controller. The controller looks into the 10 tuple packet information, determines the egress port and sends a "FlowMod" message to the switch to modify a switch flow table. The following packets in the same flow will be sent through the same egress port. Because there are two aggregation switches, i.e., two paths from N_0 to N_2, the OpenFlow controller can have two options to determine the egress port. That is, the egress port can be 1 or 2, which means the flow can go through the aggregation switch S_{A0} or the aggregation switch S_{A1}.

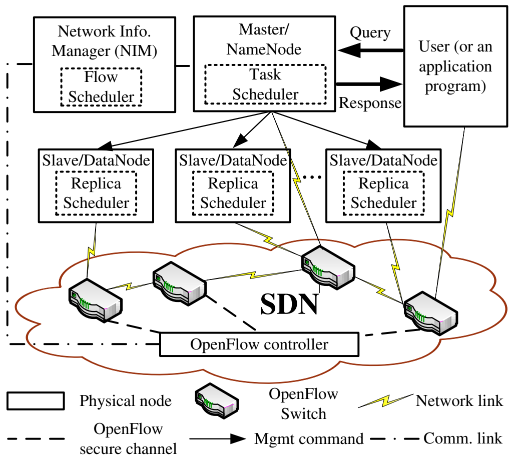

Fig. 3. Cormorant system architecture

III. Cormorant Design

In this section, we describe the system that we designed and implemented to evaluate the promise of SDN for improved Hadoop MapReduce query processing.

A. System architecture

Figure 3 shows the overall system architecture. The system is mainly comprised of Hadoop (with Master/NameNode and Slave/DataNode servers deployed in separate nodes), a network information manager, and an OpenFlow controller.

The basic operation of the system is as follows: The OpenFlow controller collects all the flow information from all the OpenFlow switches and generates a snapshot of current network status at a constant interval regularly. This information is stored at the Network Information Manager (NIM) and can be shared by the task scheduler, the replica scheduler and the flow scheduler. When Hive receives a query, it translates the query into several map reduce jobs and submits the jobs to the Hadoop master. Based on the network status snapshot, the task scheduler at the master node assigns tasks to different slaves; the replica scheduler at each slave node selects replicas; and the flow scheduler schedules the flows. After all the jobs finish, the query results are returned to the user.

Table I lists the notations for the rest of the paper.

TABLE I
NOTATIONS

Cap	port capacity (1Gbps in our setting)
N	a physical node
$Flow$	a network flow defined by 10 tuples
$\mathbf{Flow}$	a set of all the flows
P_{Flow}	a random variable that denotes a path for $Flow$
$\mathbf{P}_{Flow}$	a sample space of all candidate paths for $Flow$
p_{Flow}	a physical path in a sample space $\mathbf{P}_{Flow}$
$A(p_{Flow})$	available bandwidth of path p_{Flow}
$t \in \mathbf{Task}$	a task in a task set

B. Network Information Manager (NIM)

The NIM updates and inquires about the information on the current network state by communicating with the OpenFlow controller. The network information includes the network topology, queues, links, and their capabilities. It is important to keep this information up-to-date as inconsistency could lead to under-utilization of network resources as well as bad query performance. The NIM maintains a network status snapshot by collecting traffic information from OpenFlow switches. When a scheduler sends an inquiry to the NIM to inquire $A(p_{Flow})$, it will return the current available bandwidth of the flow by finding out the hop along the whole path that has the minimum available bandwidth (bottleneck).

Besides inquiring $A(p_{Flow})$, i.e., the available bandwidth for $Flow$ with a specific path p_{Flow}, the schedulers can also inquire a list of candidate paths. In this case, the NIM can select the best path that has the maximum $A(p_{Flow})$. Based on the best path information, the OpenFlow controller can send a "FlowMod" message to the switch to modify the switch flow table to add the best path.

C. Task/Replica scheduler

We follow the basic idea for task scheduler proposed in Mantri [3], i.e., placing a task close to its data. Compared with the default task scheduler which uses the static node-local, rack-local, and non-local tags, the improved tasks scheduler uses real-time global network status information for all the tasks. It greedily selects a task with *the most available bandwidth* from the data node to the taskTracker. Note that we assume that more available bandwidth may make the task finish faster and this approach is only "local" optimal for this task but may not be "global" optimal for all the tasks.

We apply Algorithm 1 for task sets. It picks the one that has the maximum available bandwidth (line 9). Finally, it compares the maximum available bandwidth with a threshold (a configurable system parameter). It will return the task if the maximum available bandwidth is more than the threshold and return no task if the maximum available bandwidth is less than the threshold (which means there is serious network congestion and/or there is no available slots and it may be better to postpone executing this task until the situation improves).

There is one key parameter P in Algorithm 1, which denotes an "abstract" path from N_d to N_c. It is called an "abstract" path because it is defined from a "MapReduce" point of view, which is different from a physical path that is defined from a "Network" point of view. We use a discrete random variable P to denote an abstract path and use p to denote a physical path. We use $\mathbb{P} = \{p_1, p_2, ..., p_n\}$ to denote the sample space of all the n candidate physical paths. $A(P)$ is calculated as the average of the available bandwidth of all the n candidate physical paths.

Algorithm 1: Select task from a task set

```
1  Input: taskTracker at node $N_c$ which is asking the master for a task
   to execute. task set $Task$;
2  Output: the task $t_{exec} \in Task$ to be executed;
3  $Max = -infinity; t_{exec} = null$;
4  for $t \in Task$ do
5      NodeSet $Datanode$=t.getSplitLocations();
6      for $N_d \in Datanode$ do
7          Path $P$ = $new$ Path($N_d$, $N_c$);
8          if $A(P) > Max$ then
9              $Max = A(P); t_{exec} = t$;
10         end
11     end
12 end
13 if $Max > Threshold$ then
14     Return $t_{exec}$;
15 end
16 else
17     Return null;
18 end
```

When a task is actually executed, which replicas to choose is determined by the slave. This means the replica scheduler work independently with task scheduler which could cause inconsistency. So we also modified the source code of HDFS to make them work collaboratively. When a taskTracker needs to read a chuck, it also selects the replica that has *the most available bandwidth* to the taskTracker.

D. Flow scheduler

We follow the flow scheduler design in Hedera [2], 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 then chooses a path whose available bandwidth can best accommodate this flow and schedule the flow to that path. 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.

E. Collaborative schedulers

We described three improved schedulers in the previous three subsections. However, they may not be able to deliver the best optimized performance if they work separately as opposed to working collaboratively as shown below. (1) Task/replica scheduler only. Although we select a task with *the most available bandwidth* from the data node to the taskTracker, the most available bandwidth is not guaranteed at run-time if the task scheduler does not work collaboratively with a flow

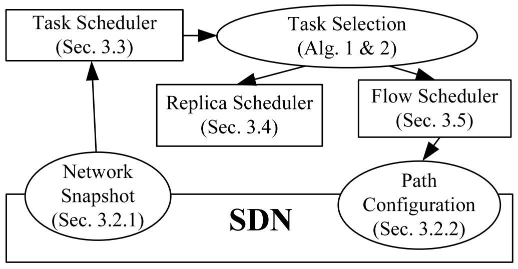
Task Scheduler (Sec. 3.3)
Task Selection (Alg. 1 & 2)
Replica Scheduler (Sec. 3.4)
Flow Scheduler (Sec. 3.5)
Network Snapshot (Sec. 3.2.1)
SDN
Path Configuration (Sec. 3.2.2)

Fig. 4. Collaborative relationship among schedulers

scheduler. (2) Flow scheduler only. If none of the candidate paths has enough available bandwidth due to neighbor traffic, the flow scheduler will not have good choice and the scheduled task may be executed slow.

In order to improve this, we build the collaborative schedulers as shown in Figure 4. (1) Network status snapshot is built by leveraging SDN. (2) The collaborative task scheduler chooses the best task with *the most available bandwidth* based on the network status and Algorithms 1. $A(P)$ is calculated as the maximum available bandwidth of all the n candidate physical paths. (3) The replica scheduler will choose the replica accordingly. (4) The flow scheduler leverages the path configuration handler and schedules the physical path that has *the maximum available bandwidth* which corresponds to the task scheduler's choice.

Note that, collaborative schedulers are not simply putting the improved task/replica/flow schedulers all together. For example, in the improved task scheduler, $A(P)$ is calculated as the average because the scheduler is uncertain about the physical path. However $A(P)$ is calculated as the maximum in the collaborative one because the collaborative task scheduler is sure about the physical path.

IV. Experimental Evaluation

A. Experimental Setup

Our test bed 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 Hexa-Core CPU, 32GB of RAM, 1TB 7200rpm disk running Linux with kernel 2.6.32. Six of the machines N_{10-15} are installed with a 4-port Gigabit NetFPGA card and perform as OpenFlow switches. Seven of the machines $N_{0-3,5-7}$ are used for Hadoop MapReduce deployment with one master (at N_0) and six slaves (at $N_{1-3,5-7}$). N_0 and N_4 are also used for generating neighbor network traffic. N_8, N_9 and N_{16} are used to run network information manager(NIM), Openflow Controller, and client emulator, respectively.

We run Hadoop MapReduce 1.2.1 and follow most of the default settings. We use Hive 0.11 and run an OLAP benchmark TPC-H with a scaling factor of 100.

We use nodes N_0 and N_4 to create 12 *neighbor contention flows* emulated by `iperf` [2]. $F_{i \to j}$ denotes a flow from node

[2]http://iperf.sourceforge.net/

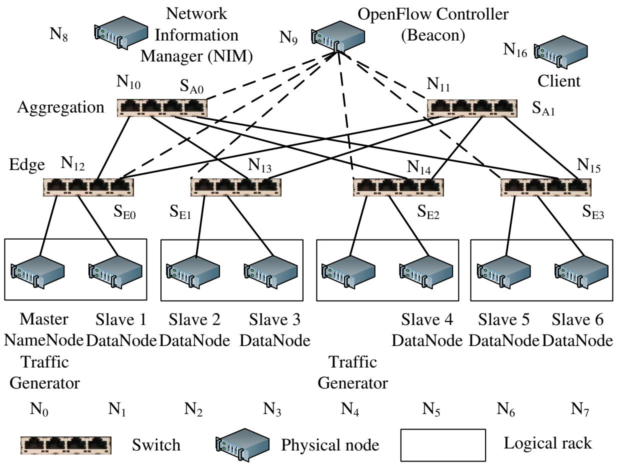

Fig. 5. Hadoop MapReduce Setup

N_i to node N_j. We have 4 large flows $F_{3\to0}$, $F_{7\to4}$, $F_{0\to2}$ and $F_{4\to6}$ with 800Mbps and 8 small flows with 50Mbps.

Each experiment is run three times and the average (with the standard deviation if applicable) is reported. In Section IV-B, we first summarize all TPC-H queries' performance under 5 different scenarios as shown in Table II, i.e., **default** (default Hadoop), **task/replica scheduler only**, **flow scheduler only**, **collaborative** and **no traffic** (default Hadoop without any neighbor traffic).

TABLE II
SCHEDULERS USED IN DIFFERENT SCENARIOS

Scenarios	Task Scheduler	Flow Scheduler	Traffic
default	default	ECMP	yes
task/rep. sched.	improved	ECMP	yes
flow sched.	default	improved	yes
collaborative	collaborative	collaborative	yes
no traffic	default	ECMP	no

B. Summary of TPC-H queries' performance

In this section, we discuss the experimental results for all TPC-H queries (Q1-Q22) as shown in Figure 7. The y-axis is the query execution time and x-axis shows 5 different scenarios for each query. Besides the other default settings of Hadoop, we set the number of replica as 1, the chunk size as 512MB, the number of map slots as 3 and the number of reduce slots as 3. The neighbor network traffic is medium, i.e., 45% network bandwidth utilization.

We summarize and compare the whole TPC-H benchmark execution time in Figure 6 and we enumerate all the details of each query execution time in Figure 7. We have the following observations: (1) Figure 6 shows that it takes on average 17757s, 16043s, 16090s, 13756s and 11191s for default, task/replica scheduler only, flow scheduler only, collaborative and no traffic scenarios. Although the performance is reduced when task/replica scheduler or flow scheduler is used, the benefit is quite limited (less than 10%). When all the schedulers work collaboratively, we can achieve benefits (22.5%) beyond simply adding up their own benefits. Since the execution time

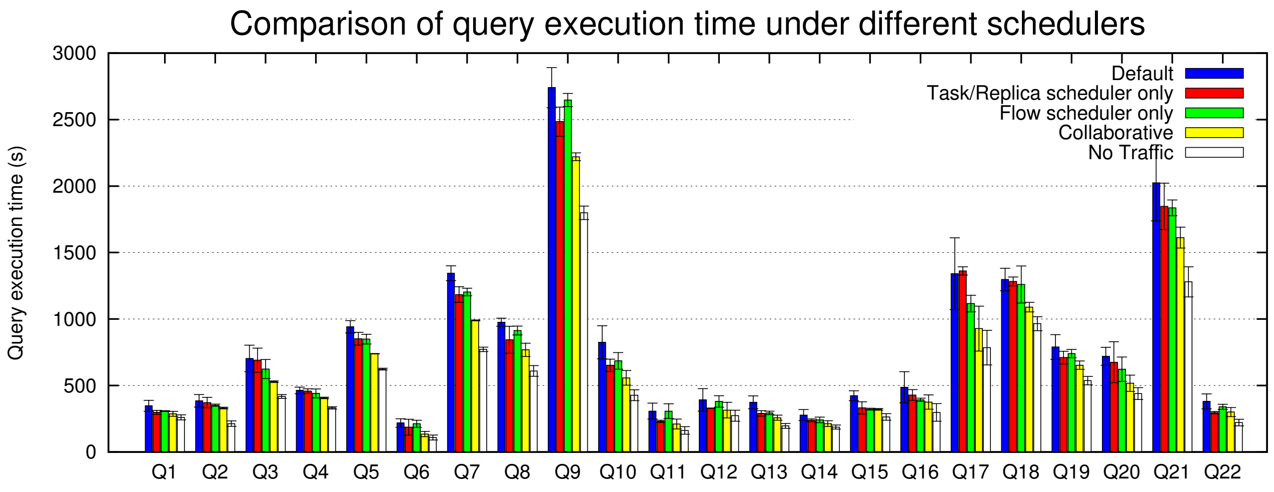

Fig. 7. Details of TPC-H benchmark query execution time

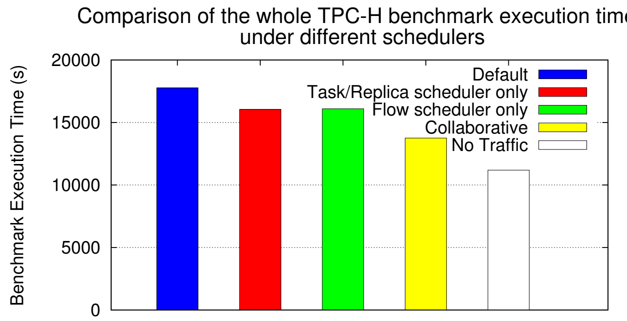

Fig. 6. Comparison of the whole TPC-H benchmark execution time

of no traffic scenario is 11191s, ideally we can improve the performance by at most 37%. In another word, Cormorant reduces 60% of the overhead brought by network traffic. (2) From Figure 7, we can see that, for different queries, different improvement is achieved. But the query execution time for every query is reduced when we use task/replica scheduler or flow scheduler alone. Across all of the benchmark queries, the collaborative case is always delivering the best performance.

V. Conclusions

In this paper, we propose Cormorant, a Hadoop-based system with collaborative software-defined networking for executing analytic queries. Unlike previous work, in which Hadoop works independently from the network underneath, our system enables Hadoop and the networking layer to work together to improve network utilization and reduce query execution times. As our experiments with an implementation show, this improvement goes beyond that achievable by the state of the art approach of combining optimizing task schedulers and flow schedulers without collaboration. We believe that our work shows early promise for achieving one of the often-cited goals of SDN, i.e., tightly integrating applications with the network to improve performance

Of course, substantial room for future work exists. For example, it might be possible to generate an optimal network bandwidth schedule for all tasks (not just those from a single application) a priori. Then a collaborative flow scheduler could enforce this optimal schedule. Also, there are new systems, such as Spark [10], that primarily use main memory storage rather than disk storage. It would be interesting to explore the performance improvements achievable by exploiting SDN in such an environment.

VI. Acknowledgments

We thank all the reviewers for their insightful comments. This work is supported in part by NSF grants CNS-1339839, CNS-1422245.

References

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proc. of SIGCOMM*, 2008.
[2] 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.
[3] 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.
[4] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proc. of OSDI*, 2004.
[5] C. Hopps. Analysis of an equal-cost multi-path algorithm. *RFC 2992, IETF*, 2000.
[6] 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.
[7] Open Networking Foundation. Software-Defined Networking: The New Norm for Networks. 2013.
[8] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha. Sharing the data center network. In *Proc. of NSDI*, 2011.
[9] P. Xiong, H. Hacigumus, and J. F. Naughton. A software-defined networking based approach for performance management of analytical queries on distributed data stores. In *Proc. of SIGMOD*, 2014.
[10] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proc. of HotCloud*, 2010.