Pronto: A Software-Defined Networking based System for Performance Management of Analytical Queries on Distributed Data Stores

Pengcheng Xiong
NEC Laboratories America
pxiong@nec-labs.com

Hakan Hacıgümüş
NEC Laboratories America
hakan@nec-labs.com

ABSTRACT

Nowadays data analytics applications are accessing more and more data from distributed data stores, creating large amount of data traffic on the network. Therefore, distributed analytic queries are prone to suffer from bad performance in terms of query execution time when they encounter a network resource contention, which is quite common in a shared network. Typical distributed query optimizers do not have a way to solve this problem because historically they have been treating the network underneath as a black-box: they are unable to monitor it, let alone to control it. However, we are entering a new era of software-defined networking (SDN), which provides visibility into and control of the network's state for the applications including distributed database systems. In this demonstration, we present a system, called Pronto that leverages the SDN capabilities for a distributed query processor to achieve performance improvement and differentiation for analytical queries. The system is the real implementation of our recently developed methods on commercial SDN products. The demonstration shows the shortcomings of a distributed query optimizer, which treats the underlying network as a black box, and the advantages of the SDN-based approach by allowing the users to selectively explore various relevant and interesting settings in a distributed query processing environment.

1. INTRODUCTION

To become more efficient, effective, and competitive, enterprises are expecting ever increasing benefits from data analytics. To meet this demand, data analytics platforms are including more data sources, which are both internally and externally available. Those data sources are typically stored in distributed data stores [1]. Data analytics applications or data scientists query the data from these distributed stores and merge and join the data to generate coherent analysis reports. With continuously increasing data sizes, querying and joining data from those distributed sources can generate a significant amount of data traffic on the network, an issue which is exacerbated if the network is shared by other applications as well. Therefore, optimizing queries that access the distributed data stores, and specifically optimizing their network utilization, is likely to be an important problem to address in order to deliver improved query performance and query service differentiation.

However, the interaction between the query optimizer and the network in prior distributed query optimization work has been limited in that the optimizer has no direct way to (1) inquire about the current status and performance of the network, and (2) control the network with directives, e.g., making bandwidth reservations, which would improve query performance management. Although some distributed query optimizers and execution systems try to react in some fashion to unexpected network conditions [4], the decisions they make are either heuristic-driven or potentially inaccurate due to a lack of ability to predict the true costs of remote data access. It is perhaps reasonable to expect that with greater visibility into the network's state, a distributed query optimizer could make more accurate cost estimates for different query plans and make better informed decisions. Moreover, it is also intuitively clear that, with some control of the network's future state, a distributed query optimizer could request and reserve the network bandwidth for a specific query plan and thereby improve query performance and query service differentiation. The problem is that historically the networks that distributed data management systems rely on have made both these tasks difficult or impossible.

Yet, we are entering a new era of software-defined networking (SDN) [2, 3]. These networks enable such visibility into and control of the network's state for a distributed query optimizer thereby addressing the key limitations we mentioned above. By decoupling the system that makes decisions about where traffic is sent (the control plane) from the underlying systems that forwards traffic to the selected destination (the data plane), network services can be managed through an abstraction of lower level functionality. SDN raises the possibility that it is for the first time feasible and practical for distributed query optimizers to carefully monitor and even control the network. We have developed a group of methods that exploit the capabilities of SDN to optimize the performance of analytical queries on distributed data stores [6], which analyzes and shows the opportunities of SDN for distributed query optimization. In this demonstration, we present the system, called *Pronto*, that is the real implementation of those methods on commercial SDN products. The demonstration shows the

Proceedings of the VLDB Endowment, Vol. 7, No. 13

shortcomings of a distributed query optimizer, which treats the underlying network as a black box, and the advantages of the SDN-based approach by allowing the users to selectively explore various relevant and interesting settings in a distributed query processing environment.

2. PRONTO SYSTEM ARCHITECTURE

Figure 1 shows the overall system architecture. The system is mainly composed of a user site, a master site, several data store sites, and an SDN component, which consists of an OpenFlow controller (Beacon [1]) and OpenFlow switches (NEC OpenFlow PFS5240). The unit of distribution in the system is a table and each table is stored at one data store or can be replicated at more than one data store sites. The user or the application program submits the query to the master site for compilation. The master site coordinates the optimization of all SQL statements. We assume that only the data store sites store the tables. The master and the data stores run an off-the-shelf, modified database server (PostgreSQL, in our case).

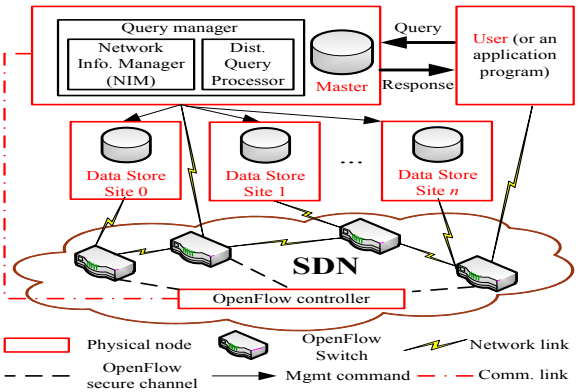

Figure 1: System architecture

We have a query manager running on the master site, which consists of a distributed query processor and a network information manager (NIM). The distributed query processor presents an SQL API to users. It also maintains a global view of meta-data of all the tables in the databases. The query manager communicates with the OpenFlow controller to (1) receive network resource usage information, and update the information in NIM accordingly; and (2) send the control commands to the OpenFlow controller.

The basic operation of the system is as follows: when the query manager receives a query, it parses the query, generates and optimizes a global query plan. The global query plan is divided into local plans. The local plans are sent to corresponding data store sites for execution via separate threads. The query manager orchestrates the necessary data flows among the data store sites. The query manager also forwards the final results from the master to the user.

3. SDN, OPENFLOW, PERFORMANCE MANAGEMENT WITH OPENFLOW APIS

In this section we give a brief overview of Software-Defined Networking (SDN), OpenFlow, and how we utilize the OpenFlow APIs in our system implementation. 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 [3]. OpenFlow is a standard communication interface among the layers of an SDN architecture, which can be thought of as an enabler for SDN [2]. 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 the packet information. More specifically, APIs in the OpenFlow switch enable us to attach the new flow to one of the physical transmitter queues behind each port of the switch.

For example, we show a commercial OpenFlow switch and three data store sites $S_{0,1,2}$ are connected to the switch at port 0,1,2 in Figure 2, respectively. When a new flow $Flow_0$(from S_0 to S_2) generated by a user arrives, a "PacketIn" message is sent from the switch to the controller. The controller looks into the packet information, determines the egress ports (i.e., 2) and one of the transmission queues (e.g., q8) according to the user's priority 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 transmission queue q8 of the egress ports (i.e., 2) to site S_2. If no user information is specified, a default queue (q4) will be used.

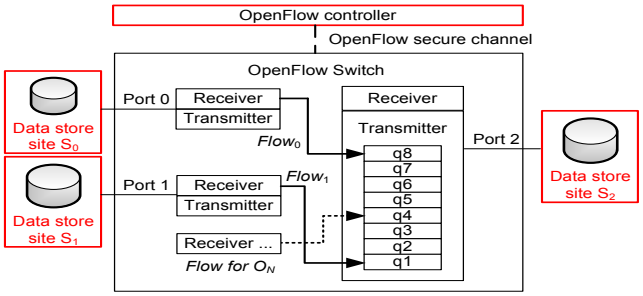

Figure 2: Inside an OpenFlow switch

From the databases point of view, the abstraction and the control API allow the database to get available network bandwidth as well as to set network traffic differentiation. (1) The distributed query processor can send an inquiry to the NIM to inquire the available bandwidth, which can be used to calculate the cost of candidate plans. (2) The distributed query processor can communicate with the NIM which contacts the OpenFlow controller to leverage the OpenFlow APIs to pro-actively notify the switch to give certain priority to or make a reservation for specific flows. The main mechanism in the OpenFlow switch to implement these methods is the transmission queues.

The first point is illustrated in the following example. Figure 3 shows the query execution time (y-axis) of two candidate plans (plan 0 and 1) for repeated execution of TPC-H Q14 in a distributed setting, where the join tables are hosted in two separate data store sites. (The more specific details of the experimental setup are not too important for illustration purposes.) The horizontal lines at 600th and 1200th seconds mark the observed network status changes. We can see that, as the network status between the sites changes, the query execution times for specific plans (Plan 0 and 1 in

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

the figure) are significantly affected. If a distributed query optimizer treats the network as a fixed cost, black box resource and sticks to a fixed plan, which is the case for most distributed database query optimizers, it may not be able to choose the plan with the shortest execution time. In our system the query optimizer communicates with NIM and chooses the appropriate plan based on the status of the network.

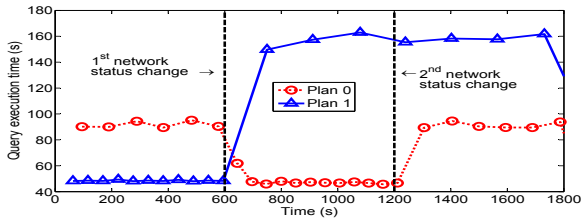

Figure 3: TPC-H Q14 different plans' execution time in a distributed setting

For the second point, i.e., priority management, when the queues within the switch are set as priority queues (PQ), the PQ sends the frames in the order of queue priority. During transmission, this system gives the higher-priority queues higher preference over the lower-priority queues. If any port is set as PQ, then the queues from the highest priority to the lowest priority are q8,q7,...,q1. When the queues within the switch are set as Weighted fair queue (WFQ), the switch sends the amount of frames equivalent to the minimum guaranteed bandwidth from each queue to begin with. Under both settings, the calculation of the available bandwidth for a certain query plan should be changed accordingly.

4. DEMONSTRATION

This demonstration will show the novel features of Pronto and also provide an intuitive way of understanding the limitations of existing systems. The demo system implements the architecture described in Section 2.

4.1 Demonstration Components

Distributed Query Processor (DQP): DQP implements the methods we referred to in Section 3. We will demonstrate DQP's capabilities to 1) adaptively choose an optimal distributed query plan based on the network status and 2) deliver differentiated query performance management through prioritization.

The DQP uses a model, which estimates the total response time for a query. This estimate is used in the optimizer's cost model. We use the framework proposed in [5] to estimate the cost for a local operator and we propose our own method to estimate the cost for a network operator. More details can be found in [6].

Command Line Interface (CLI): The system includes a CLI that allows the users to interact with the system. The valid CLI commands are similar to the ones in other commercial database products where the users can interact with and query the database optimizer (such as the *explain* facility).

Traffic Emulator: The system includes a traffic emulator that allows the users to create a network traffic with very specific characteristics to emulate realistic situations in the actual system. The traffic emulator is implemented and added to the demo system to give the users the ability to experiment with the system and explore interesting situations, which could emerge in the real life situations. Naturally when our system is running in real deployment it directly works with the actual traffic conditions created by actual applications.

4.2 Demo Scenarios

We demonstrate the following 4 scenarios: 1) We show how the users can see alternative distributed query execution plans generated by the system. 2) We show how the system adaptively chooses a dynamic plan based on the available network bandwidth monitored by SDN. 3) We show how differentiated query execution can be achieved when the user gives a specific priority to a specific workload and when the user reserves a specific bandwidth capacity for a specific workload through SDN. 4) We also show how the user can manipulate the system conditions to generate real life situations and explore the previous scenarios under varying conditions. In the following illustrations we use TPC-H Q14 as an example. The data tables that include the necessary TPC-H benchmark data are distributed over the data stores in the system.

4.2.1 *Query Plan Generation and Reporting*

Figure 4 shows how the user can query the optimizer to show the alternative plans generated for a specific query. Each plan is a tree such that each node of the tree is a physical operator. Moreover, the site that will execute the operator is labeled next to the operator. The screen shows 3 alternative plans (plan 0,1, and 2). As an example, `plan 0` performs the final aggregation in the query in server `jedi22` whereas `plan 1` performs the aggregation in server `jedi21`. This function allows the users to understand, analyze, and debug the specific details of complex distributed query execution plans.

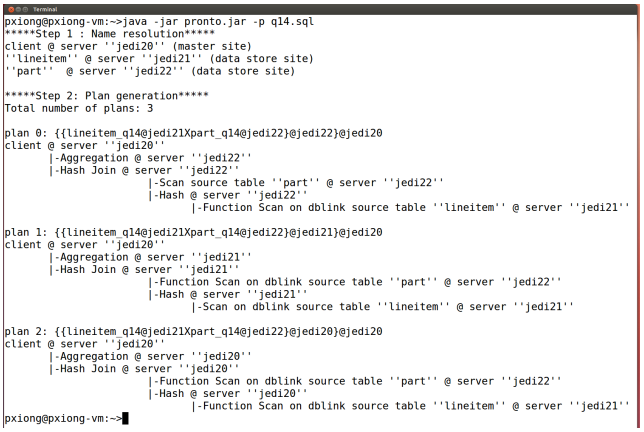

Figure 4: Query Plan List

4.2.2 *Network-Aware Query Execution*

This scenario shows the two ways to execute a query, i.e., with a default query optimizer (i.e., *network-unaware*) or with a query optimizer supported by SDN (*network-aware*). The expected behavior of the network-aware optimizer is similar to the example we gave in Section 3, where the optimal plan dynamically changes based on the network status.

The network-unaware query optimizer. When the user executes a query with a default query optimizer, the user just simply types in the query without any parameter and the results are returned with total runtime.

The SDN-enabled (network-aware) query optimizer. When the user executes a query with our SDN-enabled query

optimizer, the optimizer uses the network bandwidth monitoring and control knobs provided by SDN to adaptively choose a plan. This option is enabled by "-d" parameter as shown in Figure 5. The query optimizer supported by SDN obtains real-time network information from NIM and ranks all the plans according to their estimated runtime. The optimization time is also calculated. The best plan is chosen according to the cost model and executed. The final results are returned with total runtime.

```
pxiong@pxiong-vm:~>java -jar pronto.jar -e q14.sql -d
*****Execute q14.sql (dynamic plan)*****
plan 0: {{lineitem_q14@jedi21Xpart_q14@jedi22}@jedi22}@jedi20, Est. 45057ms

plan 1: {{lineitem_q14@jedi21Xpart_q14@jedi22}@jedi21}@jedi20, Est. 48008ms

plan 2: {{lineitem_q14@jedi21Xpart_q14@jedi22}@jedi20}@jedi20, Est. 71032ms

Optimization time: 8.702ms

*****(plan 0 is chosen)*****

*****Results*****

    promo_revenue
---------------------
 16.64035743325410 25
(1 row)

Total runtime: 47037.462 ms
pxiong@pxiong-vm:~>
```

Figure 5: Execute query with a dynamic plan

Investigation Option. Moreover, we also provide a function for the user to investigate all the candidate plans. By using "-f" parameter the user can force the optimizer to use a specific plan (the list of the plan can be obtained as described in Scenario 1 above.). For example, the user specifies plan 0 for Q14 to be executed in Figure 6. This is a very useful function for the users to investigate the performance of specific plans.

```
pxiong@pxiong-vm:~>java -jar pronto.jar -e q14.sql -f 0
*****Execute q14.sql (fixed plan)*****
plan 0: {{lineitem_q14@jedi21Xpart_q14@jedi22}@jedi22}@jedi20
client @ server ''jedi20''
       |-Aggregation @ server ''jedi22''
       |-Hash Join @ server ''jedi22''
                   |-Scan source table ''part'' @ server ''jedi22''
                   |-Hash @ server ''jedi22''
                           |-Function Scan on dblink source table ''lineitem'' @ server ''jedi21''


*****Results*****

    promo_revenue
---------------------
 16.64035743325410 25
(1 row)

Total runtime: 46057.763 ms
pxiong@pxiong-vm:~>
```

Figure 6: Execute query with a specified plan

4.2.3 *Differentiated Query Execution.*

Here we first show how the user can set a higher priority for a specific workload. For example, the user can configure the OpenFlow switch in PQ mode and specify the highest priority "q8" for the distributed data store traffic as shown in Figure 7. In this case, the dynamic query optimizer will ignore the contention traffic with lower priorities when it dynamically chooses the best plan.

```
pxiong@pxiong-vm:~>java -jar pronto.jar -pri jedi21 jedi22 8
*****SDN-based network traffic priority configuration*****
Done! Configure network traffic priority jedi21->jedi22 at q8
Total configuration time: 22.293 ms
pxiong@pxiong-vm:~>
```

Figure 7: Specify the priority for a workload

Furthermore, the user can "reserve" a certain amount of network bandwidth for a specific workload. For example, the user can configure the OpenFlow switch in WFQ mode and make a reservation of 600Mbps for a flow from a server `jedi21` to another server `jedi22` as shown in Figure 8. In this case, the dynamic query optimizer will take the bandwidth reservation into consideration when it dynamically chooses the best plan.

```
pxiong@pxiong-vm:~>java -jar pronto.jar -prov jedi21 jedi22 600
*****SDN-based network bandwidth provision*****
Done! Provision jedi21->jedi22 600Mbps
Total configuration time: 10.341 ms
pxiong@pxiong-vm:~>
```

Figure 8: Make reservation for a workload

4.2.4 *Contention traffic manipulation*

We build a tool to monitor the real-time flow rates and to manipulate contention traffic as shown in Figures 9 and 10, respectively.

```
15:25:27.057 [Timer-2] INFO  n.b.t.LearningSwitchTutorialSolution - Flow 24,27,4 : 3.3114054E7
15:25:27.058 [Timer-2] INFO  n.b.t.LearningSwitchTutorialSolution - Flow 27,24,4 : 0.0
15:25:28.057 [Timer-2] INFO  n.b.t.LearningSwitchTutorialSolution - Flow 24,27,4 : 3.364437E7
15:25:28.058 [Timer-2] INFO  n.b.t.LearningSwitchTutorialSolution - Flow 27,24,4 : 60900.0
15:25:29.056 [Timer-2] INFO  n.b.t.LearningSwitchTutorialSolution - Flow 24,27,4 : 3.2109372E7
15:25:29.057 [Timer-2] INFO  n.b.t.LearningSwitchTutorialSolution - Flow 27,24,4 : 61600.0
15:25:30.057 [Timer-2] INFO  n.b.t.LearningSwitchTutorialSolution - Flow 24,27,4 : 0.0
15:25:30.059 [Timer-2] INFO  n.b.t.LearningSwitchTutorialSolution - Flow 27,24,4 : 71260.0
15:25:31.057 [Timer-2] INFO  n.b.t.LearningSwitchTutorialSolution - Flow 24,27,4 : 3.5769988E7
15:25:31.058 [Timer-2] INFO  n.b.t.LearningSwitchTutorialSolution - Flow 27,24,4 : 55790.0
15:25:32.056 [Timer-2] INFO  n.b.t.LearningSwitchTutorialSolution - Flow 24,27,4 : 3.6938788E7
15:25:32.057 [Timer-2] INFO  n.b.t.LearningSwitchTutorialSolution - Flow 27,24,4 : 54180.0
```

Figure 9: Real-time flow rates

```
pxiong@pxiong-vm:~>java -jar pronto.jar -traffic jedi21 jedi22 900 3600
*****Generate traffic*****
Done! Generate ''900Mbps'' traffic from ''jedi21'' to ''jedi22'' for ''3600''s
Total generation time: 5.834 ms
pxiong@pxiong-vm:~>
```

Figure 10: Generate contention traffic

For example, we can observe that the real-time rate of a flow from port 24 ("jedi24") to port 27 ("jedi27") on queue 4 is around 33MBps from the first line in Figure 9. As another example, we can generate a contention network traffic at the rate of 900Mbps from server "jedi21" to server "jedi22" as shown in Figure 10. In this case, we can compare the query execution times with the default (network-unaware) query optimizer and the dynamic (network-aware) query optimizer supported by SDN. We would see that the default query optimizer chooses the query Plan 0, which exhibits a quite long query execution time due to the network contention whereas the dynamic query optimizer adaptively chooses Plan 1, which has a much shorter execution time than Plan 0, by leveraging the network status information from SDN.

5. REFERENCES

[1] D. Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4), Dec. 2000.

[2] 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.

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

[4] T. Urhan, M. J. Franklin, and L. Amsaleg. Cost-based query scrambling for initial delays. In *Proc. of SIGMOD*, 1998.

[5] W. Wu, Y. Chi, S. Zhu, J. Tatemura, H. Hacıgümüş, and J. F. Naughton. Predicting query execution time: Are optimizer cost models really unusable? In *Proc. of ICDE*, 2013.

[6] P. Xiong, H. Hacıgümüş, 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.