Project outline – Distributed Systems

Shumo Chu and Dominik Moritz

1 Background

Database system need to transform user's high level queries, eg. SQL, Datalog into a physical query plan for the execution. The physical query plan is the actual execution map of a database system. Usually it can be viewed as a *DAG* which consists of basic operators, such as *JOIN*, *GROUP BY*, *SCAN*, *APPLY* and relational tables. In distributed database systems, the introduction of the *SHUFFLE* operator and data partitioning makes query plans more complicated.

Myria is a distributed big data management system currently being developed in the database group. Myria aims towards building a distributed database platform to provide *big data management and analytics as a service* primarily for scientific applications.

2 Objective

This outline describes out project that will help us to understand the execution of physical query plans in the Myria database. It will allow us to efficiently debug and improve query execution.

We are developing a system that will allow us to collect information about the execution of a physical query plan in the distributed system. The data has to be collected on one node and then aggregated to provide the foundation for analyzing the data and creating a meaningful visualization. We also plan to analyze the query execution and find problems such as data skew. The visualization will be embedded in a web application.

Questions that we want to answer with this project:

1. What are the data dependencies between operators in workers?
2. What is the bottleneck of the execution? Improving which part could improve the performance the most?
3. Is there a slow worker during a certain stage of the execution that prevent the progress of the whole query plan?
4. How does network and computation contribute to the total running time? We make two hypothesis, respectively. If we assume the network has infinite bandwidth and no latency, how would the running time change. If we assume that we have infinite computation power, how would the running time change.

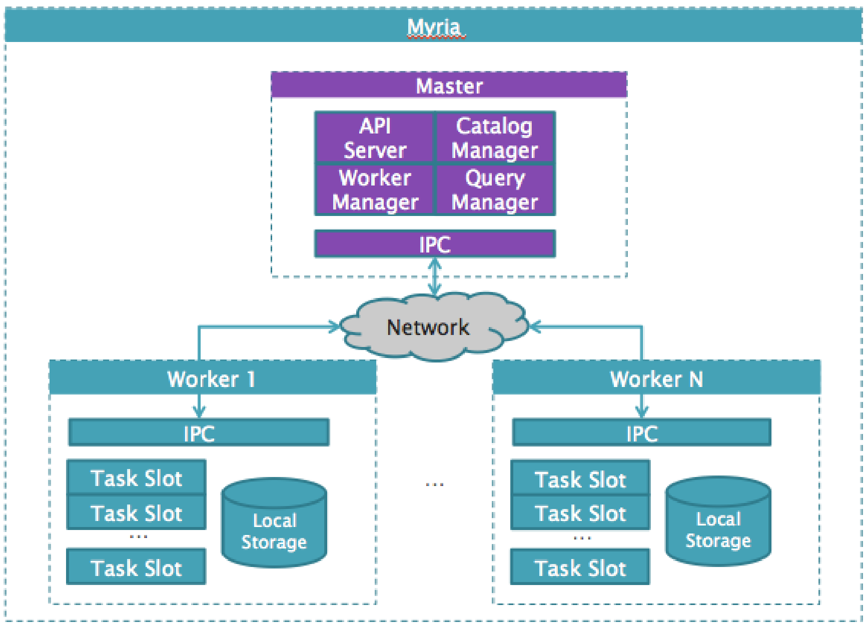

Figure 1: distributed database architecture

3 Methodoloy

3.1 Problem formulation

Database system profiling has been extensively studied over decades. There are two types of works on this topic. One is logging query execution information and analyzing logs afterwards for better query optimization. The other is using real-time profiling information to estimate the progress of a query. We focus on the first type of problem in the context of distributed database system.

Distributed database system. is a database system deployed on a cluster of servers. Figure 1 shows the system architecture of the distributed database system we use. There is a master server who parses the query and distribute the tasks the each computing node (worker). The data is also pre-partitioned and stored in each work. The database query in the form of SQL or Datalog will be translated to the physical query plan and then be executed.

Query Plan is the actually execution map for the distributed database system. Each query plan consists of basic operators forming an execution flow. For example, the SQL query $SELECT\ R.*, S.*\ WHERE\ R.x = S.y$; can be translated to the visualized query plan shown in Figure 2. The semantics of the query plan is a distributed hash join of two tables, R and S. We first need to hash each table according to the joined fields. Then tuples are shuffled by their hash results to different workers. At each worker, the shuffled tuples are joined locally. In each query plan, a query fragment is a subtree that within the boundary of network communication.

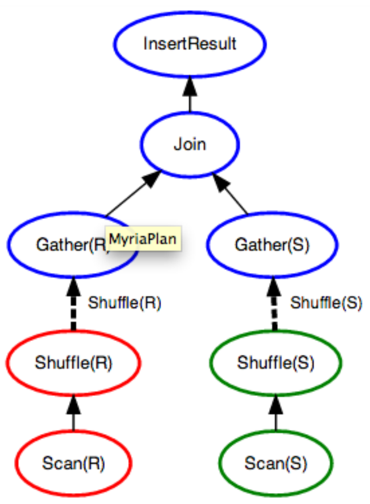

Figure 2: query plan: distributed hash join

3.2 Proposed implementation

3.2.1 Logging

In our visualization, we want to capture the state of operators, query fragments, and workers. The state of a query fragment depends on the state of the operators inside it and the state of a worker depends on the state of the query fragments that are running on it. Consequently, we only need to capture the state of query fragments per worker. In order to capture state changes, we log events and state changes of operators. In the logs we look for events that indicate a state change and signals. We use the term *state* to describe the state of an operator and the term *signal* to describe important events that happen at a certain point in time. For each state we want to visualize when the transitions happened.

Not all operators have the same states. A consumer for example does not have a wait, compute or send state. Table 1 shows the different states operators can be in and briefly describes their meaning.

In order to capture these states we have to log events that indicate their beginning and their end. An operator can only be in one state at a time. Consequently, the states can be reconstructed from the logs without any information about which events belong to each other. Since we also log the time of the events we can reconstruct when an operator changes its state and determine how long it has been in a certain state.

No all transitions between states are possible. For example, the wait state can only be reached from the compute state. A way to interpret this is that wait and compute are a sub-states of another state work. This is shown in Figure 3. This interpretation is useful since a set of computes and waits can be grouped together because they were triggered by a call to `fetchNextReady` and end with a return from the function. We will use this information in the visualization of operator states.

To be able to reconstruct the states mentioned in Table 1 we have to log all events that cause

state	operator type		
	producers	consumers	operations (*JOIN*, *MERGE*, *SCAN*, *APPLY*, ...)
receive	-	receiving data from producer over network	-
wait	child is producing	-	waiting for child to return
compute	hashing data	-	in `fetchNextReady` and not waiting for any data
sleep	has data for consumer that is not ready	no data in producer	waiting for signal
send	sending data to consumer over network	-	-

Table 1: Possible states of operators and their meaning.

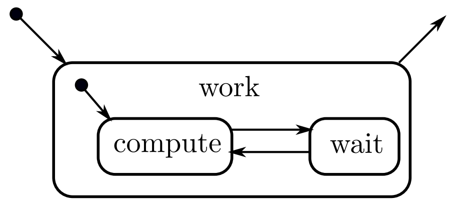

Figure 3: Extended automaton for alternative interpretation where the wait state and the compute state are sub states of a work state.

state changes. Furthermore, we will log signals.

As mentioned in the beginning of this section, the states of query fragments and workers can be inferred from the state of the query fragments that are executed on a worker. A query fragment can be in the state receiving when the leave operator is a consumer (and not a scan). It can be in the state compute when any of its children are computing. It can be sleeping when the root is sleeping and it can be sending when the root is sending. Since a fragment is executed by one thread, it cannot be in multiple states.

The state of a worker is ambiguous because multiple threads can be executing multiple query fragments that are in different states. We will use the state of the query fragments to show how many query fragments are working but we will not determine a single state for the whole worker.

For logging, we use standard Java logging using *slf4j*[1], a logging API which is implemented by different logging frameworks. Since logs are collected on different workers, we need to collect them on one machine that analyzes the data and creates the visualizations. We plan to use *Apache Flume*[2] to collect the logs from the workers. Since the collecting server is a Myria worker, we can use its database to store the logs and then read from when we create the visualization. In order to do that, we will have to implement a custom Flume sink.

[1]`http://www.slf4j.org`
[2]`https://flume.apache.org`

3.2.2 Visualization

We will use a gantt chart to visualize the states and the times an operator or query fragment is in a state. Gantt charts are used widely in the visualization of project plans because they visualize time spans and show dependencies between them. For the visualization of operators we want to show the hierarchical relationship between operators and allow users to focus only on certain operators.

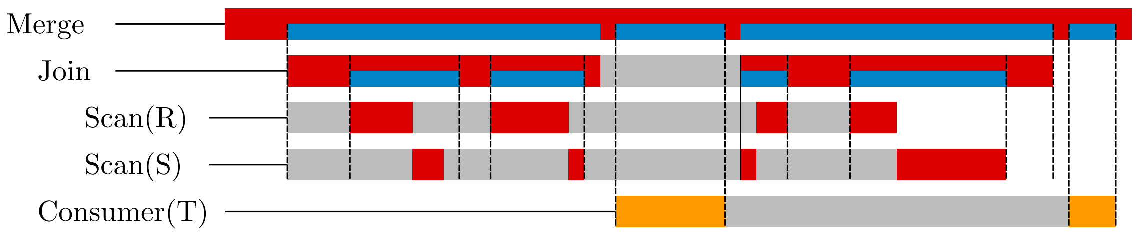

Figure 4: A prototype gantt chart. Red is computing, blue is waiting, gray is sleeping, and yellow is receiving.

d3[3] will be used to build the custom visualization. D3 is a JavaScript framework for data visualizations on the web. Consequently, our visualization will be browser based and we plan to integrate it in the existing web interface for Myria[4]. Figure 4 shows a mock-up of what the visualization of operators in a query fragment will look like. Additional information about states (such as where the data is fetched from, specific parameters of an operator and eventually how much data was returned) will be shown on demand.

This kind of visualization allows developers to answer the questions from Section 2 except for the last one. We will describe how we answer this question in Section 3.2.3. All in all, the visualization will allow us as developers to understand, debug and eventually improve how queries are executed

3.2.3 Estimate computation/network overhead

A further question is how can we get insightful information from the logging and visualization. To help us improving query plan rewriting and efficiently allocate resource for a query, we might want to reason about what factor contribute most to the running time of the query plan from the logging/visualization. In a pipelined and asynchronous query execution, since communication (via network) and computation are overlapped. There would be no accurate measure of the communication time and computation time for the whole system. So we would like to ask following questions.

What if the network in infinitely fast. That is to assume that we can have a infinite bandwidth and zero latency underlying network. Based on the logging, we want to infer that how much could the running time be improved given this assumption. This would be a measure of how expensive the computation is and could demonstrate the best performance gain we can have if we make the communication faster.

What if the computation in infinitely fast . That is to assume that we can have infinitely fast

[3] `http://d3js.org`

[4] a demo can be found at `http://myria-web.appspot.com`

computation hardware. Given this assumption, we want to infer that how much could the running time be improved. This would be a measure of how expensive the communication is.

Proposed solution. Based on the logging information, we can build a temporal state transition graph of the query execution. This graph will consist the temporal state information of each query fragment in each worker. And combined with the query plan which contains the dependency between different operators. We want to develop algorithms to infer the running time given different assumptions.