

# Investigation in Query System Framework for High Energy Physics

Thanat Jatuphattharachat, Computer Engineering, Chulalongkorn University, Thailand  
CERN, Geneva, Switzerland

## Abstract

We summarize an investigation in query system framework for HEP (High Energy Physics). Our work was an investigation on distributed server part of Femtocode, which is a query language that provides the ability for physicists to make plots and other aggregations in real-time. To make the system more robust and capable of processing large amount of data quickly, it is necessary to deploy the system on a redundant and distributed computing cluster. This project aims to investigate third party coordination and resource management frameworks which fit into the design of real-time distributed query system. Zookeeper, Mesos and Marathon are the main frameworks for this investigation. The results indicate that Zookeeper is good for job coordinator and job tracking as it provides robust, fast, simple and transparent read and write process for all connecting client across distributed Zookeeper server. Furthermore, it also supports high availability access and consistency guarantee within specific time bound.

## 1 Introduction

Femtocode is a prototype of a new query language for High Energy Physics that aims to provide the ability to make analysis plots in real-time. Unlike other real-time database system such as Impala, Femtocode supports nested data formats common in HEP. To make this query language servers data at Terabytes or Petabytes scale, the application has to be run on top of redundant commodity cluster. However, building a distributed application from scratch is difficult. One of the hardest part of distributed system is system consistency across the computing resource. Therefore, to make application work well together from every distributed part, it is necessary to implement a well-tested system which can guarantee the consistency of the system.

While a common approach is to implement every part of the system from scratch, it is better to gain the benefit from well tested third party framework. This is the subject of our investigation. This document contains of six sections. Section 2 describes the distributed Query Concept. In section 3, query system layout is explained. Section 4 shows the result of the Implementation one the local machine. The conclusion of the investigation is in the section 5, and section 6 presents possible future work.

## 2 Distributed Query Concept



Figure 1 - Query distribution

Queries come into the system in the form of user requests. For example, to process a particular data set with an analysis code, and to provide a specific output. Once the query is submitted, it may be divided into smaller pieces, or subtasks that define units of work. Subtasks can be identified uniquely using dataset and group\_id which is the number specifying subtasks group that it belongs to as a pair of keys. Subtasks then will be carried out by a single compute node. When a compute node finish executing a subtask, it will return a “partial” result to the client without waiting for other subtasks from the same query to be successfully done. This way, the user can see the real-time results executed from the system.

## 3 Query System Layout and Workflow

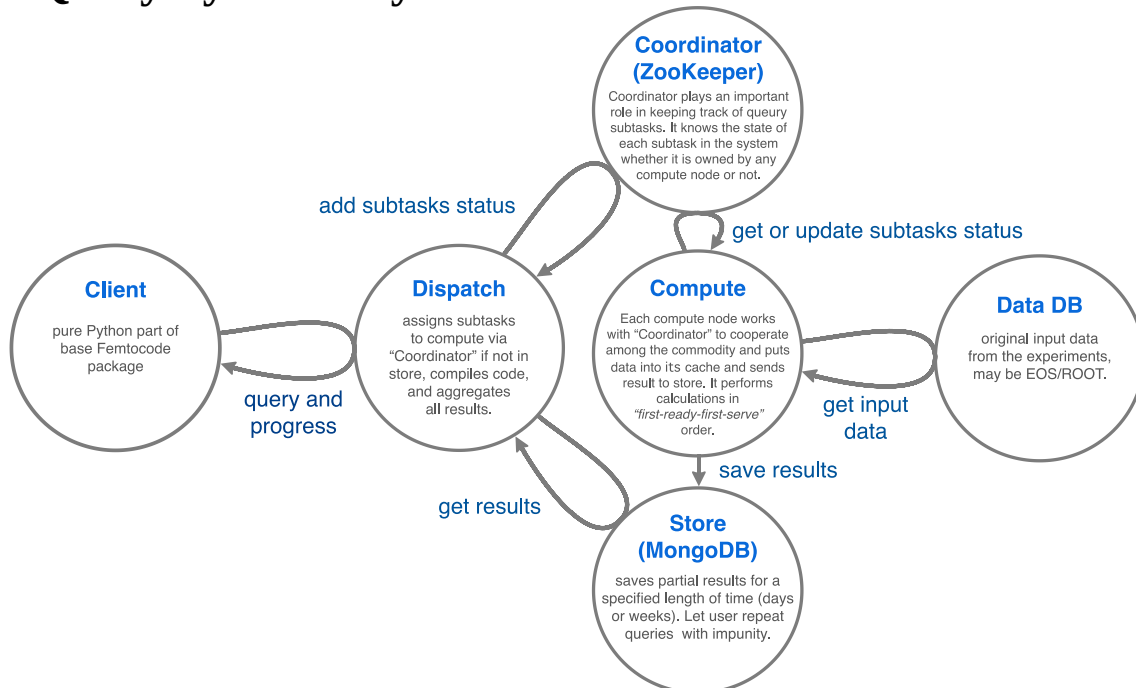


Figure 2 - shows the layout and workflow of our query system

The following sections describe each component including its function and communications.

### 3.1 Step1: Job submitted

The “Client” is the part of our system which is directly connected to the user. User can submit the job using femtocode to submit the job to the server via HTTP/HTTPS protocol. After submitting the job, the Client will check for the progress and wait for the result.

### 3.2 Step2: Job Assignment

“Job Dispatcher” serves as the component managing where each job should go. Dispatcher has the responsibility to check if the incoming job, which is identified by a pair of dataset and group id as mentioned in section 3.6, is being compiled, executed or already complete and archived in the storage or not. If this job is done by some compute node in some way, the dispatcher should wait for the job to be done and then return the partial result to the client or user. If the job has already been executed and stored in the “Store”, then Dispatcher can return the partial result to the Client immediately. However, if the job is newly submitted in to the system or might be flushed out of the storage, then Dispatcher has to connect to Job Coordinator and put the job as an “unowned” job to the coordinator so that any available computes node can get the job to executed.

### 3.3 Coordinator (Zookeeper)

Job Coordinator is responsible for keeping track of the query subtasks status and letting the other parts of the system access subtasks job status. From the investigation, Zookeeper [1] was found that it could be used as a good job coordinator. Each job (subtask) in the system will be put into Zookeeper to make available for every node in the cluster. The data stored in the “Coordinator” is only the meta data which consists of dataset, groupid, responsible worker, date-time and status of the job. The jobs tracked in the Coordinator will be divided three types, which are “unowned”, “owned” and “done”. The hierarchical namespace in Zookeeper allows the data to be divided into sub categories. Therefore, it is easier for each part of the system to come into get the data from a specific part.

### 3.4 Compute

Each “Compute” element is a node the pools of redundant commodity cluster which is the executor of the system. The main responsibility is to get the job and calculate the job as described. Each node in compute node will get the job description and job status from Zookeeper. When it checks for the job, it may wait for some time if the dataset is not in its local cache and let other compute node with the cache get the data. However, if no compute node take the job after a specific time bound it will claim the right to execute the job. This architecture makes the rebalancing of data across the computing cluster to meet the demands of popular data sets automatic. In this step, it is necessary for the compute node to change the job from “unowned” path and update the data in the “owned path”.

When the compute node gets a job to execute, it may be able to calculate the result from the cache in its local machine. However, if the dataset is new for the compute node, it must fetch the data from CMS Collaboration’s Analysis Object Data which is the result of particle collision experiments. The data format might be ROOT or some other format. When the data is ready for the compute node to execute, it will do that job until it completes the calculation. After finish executing the job, the compute node is still responsible for updating the status in the Coordinator by moving the job from “owned” to “done” path so that Dispatcher’s watcher will be able to know the progress.

### 3.5 DataDB

“DataDB” represents CMS Collaboration’s Analysis Object Data which is gathered from particle collision experiment at CERN. The data is stored in a set called dataset. DataDB is the storage which is connected to the compute cluster with internal network which can serve data transmission up to 10GB/s. Compute jobs access only local data, so DataDB is used only to refresh caches when required.

### 3.6 Store

“Store” is the facility to keep the partial results from the calculation of compute node. The result is also retrieved using a pair of dataset and groupid. The store is implement with MongoDB as a main data storage because of its ability to scale elastically.

## 4 Implementation Results

### 4.1 Phase I: Local Machine Development

In this phase, multi-threading was used to simulate the distributed and parallel programming. There were four dispatchers and four executors in the system. There was no physical data loading from CERN Collaborations Analysis Data Object. Instead simplified tasks with just a time delay were used in the system to simulate the data loading bottle neck. The language for the implementation was Python. MongoDB was launched via Docker in the local machine and was connected to Python using “Pymongo” package. Zookeeper is also installed in the local machine and integrated with Python using community supported library named “Kazoo”. There is no User Interface for job submitting because using script can populate more input than using UI. Also, it is more convenient to manage the amount of input data using the script. A test file is created and is used to check successfully created and returned jobs to ensure that the system run correctly. Several examples of running result can be found in the following figures.

```
(Thread-3 ) Received response(xid=150): {}
(Thread-3 ) Sending request(xid=151): Create(path='/unowned/entry-100-3:3-',
data='{ "state": 1, "groupid": 3, "dataset": 3}', acl=[ACL(perms=31, acl_list=['ALL'],
id=Id(scheme='world', id='anyone'))], flags=2)
(Thread-3 ) Sending request(xid=152): Create(path='/unowned/entry-100-2:2-',
data='{ "state": 1, "groupid": 2, "dataset": 2}', acl=[ACL(perms=31, acl_list=['ALL'],
id=Id(scheme='world', id='anyone'))], flags=2)
(Thread-3 ) Sending request(xid=153): GetChildren(path='/unowned', watcher=None)
(Thread-3 ) Sending request(xid=154): GetChildren(path='/unowned', watcher=None)
(Thread-3 ) Sending request(xid=155): GetChildren(path='/owned', watcher=None)
(Thread-3 ) Received response(xid=151): u'/unowned/entry-100-3:3-0000000000'
(Thread-3 ) Received response(xid=152): u'/unowned/entry-100-2:2-0000000001'
(Thread-3 ) Received response(xid=153): [u'entry-100-3:3-0000000000', u'entry-100-2:2-0000000001']
(dispatcher-worker) putting job 1:1 into zookeeper
(Thread-3 ) Received response(xid=154): [u'entry-100-3:3-0000000000', u'entry-100-2:2-0000000001']
```

Figure 3 - An example showing a dispatcher putting new job in the Coordinator log

```

(Thread-3 ) Sending request(xid=172): Create(path='/owned/entry-100-3:3-',
data='{ "state": 3, "worker": "819dee4c-1806-403c-8fcb-0b4f8e4b6e2c", "groupid": 3, "dataset": 3}',
acl=[ACL(perms=31, acl_list=['ALL'], id=Id(scheme='world', id='anyone'))]), flags=2)
(Thread-3 ) Received response(xid=172): u'/owned/entry-100-3:3-0000000000'
(Thread-3 ) Sending request(xid=173): GetChildren(path='/unowned', watcher=None)
(Thread-3 ) Received response(xid=173): [u'entry-100-1:1-0000000002', u'entry-100-2:2-0000000001', u'entry-100-4:4-0000000004', u
(Thread-5 ) waiting 100 ms for available thread
(Thread-3 ) Sending request(xid=174): GetChildren(path='/owned', watcher=None)
(Thread-3 ) Received response(xid=174): [u'entry-100-3:3-0000000000']
(Thread-3 ) Sending request(xid=175): GetChildren(path='/unowned', watcher=None)
(Thread-3 ) Received response(xid=175): [u'entry-100-1:1-0000000002', u'entry-100-2:2-0000000001', u'entry-100-4:4-0000000004', u
(dispatcher-worker) waiting 100ms for 3:3 to be executed
(Thread-6 ) waiting 100 ms for available watcher
(Thread-3 ) Sending request(xid=176): GetChildren(path='/done', watcher=None)
(Thread-3 ) Received response(xid=176): []
(watcher-worker) No data to return
(dispatcher-worker) waiting 100ms for 0:0 to be executed
(dispatcher-worker) waiting 100ms for 2:2 to be executed
(dispatcher-worker) waiting 100ms for 1:1 to be executed
(Thread-4 ) waiting for a new job that satisfy
(Thread-3 ) Sending request(xid=177): GetChildren(path='/unowned', watcher=None)
(Thread-3 ) Received response(xid=177): [u'entry-100-1:1-0000000002', u'entry-100-2:2-0000000001', u'entry-100-4:4-0000000004', u
(Thread-3 ) Sending request(xid=178): GetData(path='/unowned/entry-100-2:2-0000000001', watcher=None)
(Thread-3 ) Received response(xid=178): ({ "state": 1, "groupid": 2, "dataset": 2}, ZnodeStat(czxid=6834, mxid=6834, ctime=1502
(Thread-3 ) Sending request(xid=179): Delete(path='/unowned/entry-100-2:2-0000000001', version=-1)
(Thread-3 ) Received response(xid=179): True
(Thread-3 ) Sending request(xid=180): Create(path='/owned/entry-100-2:2-',
data='{ "state": 3, "worker": "819dee4c-1806-403c-8fcb-0b4f8e4b6e2c", "groupid": 2, "dataset": 2}',
acl=[ACL(perms=31, acl_list=['ALL'], id=Id(scheme='world', id='anyone'))]), flags=2)

```

Figure 4 - An example showing compute node owning the job log

```

(Thread-3 ) Received response(xid=189): ({ "state": 3, "worker": "819dee4c-1806-403c-8fcb-0b4f8e4b6e2c", "groupid": 3,
(Thread-3 ) Sending request(xid=190): Create(path='/done/entry-100-3:3-',
data='{ "state": 4, "worker": "819dee4c-1806-403c-8fcb-0b4f8e4b6e2c", "groupid": 3, "dataset": 3}',
acl=[ACL(perms=31, acl_list=['ALL'], id=Id(scheme='world', id='anyone'))]), flags=2)
(Thread-3 ) Received response(xid=190): u'/done/entry-100-3:3-0000000000'
(Thread-4 ) waiting for a new job that satisfy
(Thread-3 ) Sending request(xid=191): GetChildren(path='/unowned', watcher=None)
(Thread-3 ) Received response(xid=191): [u'entry-100-4:4-0000000004', u'entry-100-0:0-0000000003']
(Thread-3 ) Sending request(xid=192): GetData(path='/unowned/entry-100-0:0-0000000003', watcher=None)
(Thread-3 ) Received response(xid=192): ({ "state": 1, "groupid": 0, "dataset": 0}, ZnodeStat(czxid=6836, mxid=6836,
(Thread-3 ) Sending request(xid=193): Delete(path='/unowned/entry-100-0:0-0000000003', version=-1)
(Thread-3 ) Received response(xid=193): True
(Thread-3 ) Sending request(xid=194): Create(path='/owned/entry-100-0:0-',
data='{ "state": 3, "worker": "819dee4c-1806-403c-8fcb-0b4f8e4b6e2c", "groupid": 0, "dataset": 0}',
acl=[ACL(perms=31, acl_list=['ALL'], id=Id(scheme='world', id='anyone'))]), flags=2)
(Thread-3 ) Received response(xid=194): u'/owned/entry-100-0:0-0000000003'
(worker ) finish saving job_id:5986e40d64b4385ee6b9984b in mongo
(Thread-3 ) Sending request(xid=195): Delete(path=u'/owned/entry-100-3:3-0000000000', version=-1)
(Thread-3 ) Received response(xid=195): True
(Thread-5 ) waiting 100 ms for available thread
(dispatcher-worker) return executed 3:3 objectid:5986e40d64b4385ee6b9984b after waiting
(Thread-6 ) waiting 100 ms for available watcher
(Thread-3 ) Sending request(xid=196): GetChildren(path='/done', watcher=None)
(Thread-3 ) Received response(xid=196): [u'entry-100-3:3-0000000000']
(Thread-3 ) Sending request(xid=197): GetData(path='/done/entry-100-3:3-0000000000', watcher=None)
(Thread-3 ) Received response(xid=197): ({ "state": 4, "worker": "819dee4c-1806-403c-8fcb-0b4f8e4b6e2c", "groupid": 3,
(Thread-3 ) Sending request(xid=198): Delete(path='/done/entry-100-3:3-0000000000', version=-1)

```

Figure 5 - An example showing compute node update successfully done job log

The result shows that Zookeeper can serve the system as a job coordinator perfectly. The data in Zookeeper was persist until the next update of the data. Each node could get the same up to date data within the time scale of the speed of local machine. MongoDB is also a good solution for the permanent storage in the system due to its speed of querying. The bottle neck of the system is the execution phase. However, the execution time was just a delay in human time scale which should be replaced with a real data fetching method in the future.

## 4.2 Phase II: CERN OpenStack private Cloud Development

In this phase, virtual machines were used to test the distributed query and system high availability. CERN OpenStack Cloud with CentOS 7 were used as tested machines. Zookeeper was launched with multiple instance to test the performance of writing. The script using for this project remained the same as the previous Phase. Thus, Python, Kazoo [2], Docker [3], MongoDB [4] were all used in the system in this phase. This development phase was a part of future working plan to deploy the system in the Marathon cluster to test the system at large scale.

## 4.3 Code Reference

All the code of the project could be found at <https://github.com/JThanat/femto-mesos>

## 5 Conclusion

The aim of this project is to investigate on the query system third party framework which can satisfy the design and workflow of a distributed query system. After investigating on Zookeeper, Mesos [5] and Marathon [6], Zookeeper was selected as a job coordinator of the distributed query system. It is used to implement a high availability coordinator with key value storage which has dataset and groupid as a key and job description and the status as the value stored in a shared hierarchical namespace of Zookeeper. All the client can connect to Zookeeper and read the most up to date job status, also the client can write the data to Zookeeper without any conflict. Nevertheless, further system testing on a large-scale production environment is still needed for production performance.

## 6 Future Work

To make the system be able to deploy in production environment, it is necessary to test the system on large scale system. This may be done using Marathon or other resource management and orchestration tools depending on the existing facility infrastructure. Also, it has to be integrated with the query language part to actually run the system. Finally, MongoDB should be designed with sharing to support elastic horizontal scale when the data grow to Terabyte or Petabyte scale.

## References

- [1] Apache, "Zookeeper," 10 08 2014. [Online]. Available: <https://zookeeper.apache.org/doc/trunk/>. [Accessed 18 07 2017].
- [2] Kazoo, "Kazoo," python-zk, 14 06 2017. [Online]. Available: <https://kazoo.readthedocs.io/en/latest/>. [Accessed 18 07 2017].
- [3] Docker, "Docker," Docker Inc., 18 01 2017. [Online]. Available: <https://www.docker.com/>. [Accessed 20 07 2017].
- [4] MongoDB, "MongoDB," MongoDB, Inc., 05 07 2017. [Online]. Available: <https://www.mongodb.com/>. [Accessed 21 07 2017].
- [5] Apache, "Apache Mesos," The Apache Software Foundation, 25 05 2017. [Online]. Available: <http://mesos.apache.org/>. [Accessed 19 06 2017].
- [6] Mesosphere, "Marathon," Mesosphere, Inc., 15 06 2017. [Online]. Available: <https://mesosphere.github.io/marathon/>. [Accessed 04 07 2017].