Implementation of a Tez-like In-memory Cluster Computing Framework

Hongsen He, Xiayi Huang, Tanqing Meng
Department of Computer Sciences and Engineering
Santa Clara University
September 01 2014

Table of Contents

2 INTRODUCTION

2.1 Objective

For many resource-intensive applications, the simplest way to achieve scalable performance is to exploit data parallelism. During the past decades, a great deal of work has been created in big data and distributed computing which discover and exploit parallelism, for example, Google's Mapreduce system which has demonstrated great success allowing huge numbers of developers to be able to write concurrent software that is reliably executed in a distributed framework.

Over the years, MapReduce has served masterfully as the data processing backbone for hadoop. It is batch-oriented and cannot support interactive applications and fast processing of iterative applications. Tez represents an alternate solution to traditional MapReduce which provides fast response time and extreme throughput to meet growing standards and demands for jobs.

Based on Dryad, we are trying to implement a Tez-like framework to provide another framework which can store and analyze large volumes and variety of data efficiently to meet the evolving needs of Hadoop users. And we wish to build the new data processing framework based on YARN to meet the next phase development of Hadoop distributed file system.

On the other hand, there are other higher-level data processing applications like Hive and Pig which allow complex query logic in an effective manner with high performance. Our framework should also support this.

2.2 What is the Problem

Cluster computing frameworks like MapReduce has been widely adopted for large-scale data analytics. These systems allow users write parallel computations using a set of high-level operators, without having to worry about work distribution and fault tolerance. Although current frameworks provide numerous abstractions for accessing a cluster's computational resources, they lack abstractions for leveraging distributed memory which makes them inefficient for an important class of emerging applications: reuse of data and intermediate results across multiple computations.

Data reuse is common in many iterative machine learning and graph algorithms including PageRank, K-means clustering and logistic regression. Another important use case is interactive data mining which allow a user program run multiple ad-hoc queries on the same subset of the data. However, in most current frameworks, the only way to reuse data and immediate results is to write it to an external stable storage system, for example, distributed file system which is inefficient because it incurs substantial overheads due to data replication, disk I/O, and serialization which will slow down application execution time.

2.3 Why This is a Project Related to the This Class

Distributed file system and in-memory cluster computing provides a foundation in our

framework to support iterative and interactive applications. There are several approaches to clustering, like parallel file system or distributed file system, most of which provide features like location-independent addressing and redundancy which improve reliability or reduce the complexity of the other parts of the cluster. From which, parallel file system are a type of clustered file system that spread data across multiple storage nodes, while distributed file systems do not share block level access to the same storage but use a network protocol. Distributed file systems are commonly known as network file systems. The difference between a distributed file system and a distributed data store is that a distributed system allows files to be accessed using the same interfaces and semantics as local files.

A computer program that runs in a distributed system is called a distributed program. A distributed system or a cloud computing system may have a common goal, such as solving a large computational problem. They both are formed of groups of networked computers, which have the same goal for their work. As a matter of fact, the terms "parallel computing" and "distributed computing" have a lot of overlap. The difference between parallel computing and distributed computing is that in parallel computing, all processors may have access to a shared memory to exchange information between processors, while in distributed computing, each processor has its own private memory and information is exchanged by passing messages between the processors.

Since our framework is based on distributed file system. High-performance distributed computing in a shared-memory with the coordination of a large-scale distributed system plays an important role in our program.

2.4 Why Other Approach is No Good

The introduction of Dryad distributed data parallel programs provides a general-purpose distributed execution engine for coarse-grain data-parallel applications. It is a logical computation graph that is automatically mapped onto physical resources during the runtime. In particular, there may be many more vertices in the graph than execution cores in the computing cluster. The overall structure of a Dryad job is determined by its communication flow.

There are certain limitations of Dryad framework. One limitation of Dryad is that the job manager described in Dryad assumes that is has exclusive control of all of the computers in the cluster which is not practical in reality and it makes it difficult to efficiently run more than one job at a time. Another limitation is the inefficiency use of a shared cluster. Since the execution of a large Dryad job generates statistics on the resource usage of thousand of executions of the same program on different input data, these statistics should be reused to detect and analyzed to look for predictions of patterns. The third limitation of Dryad is the simplicity of scheduler and fault-tolerance model come from the assumption that vertices are deterministic. if an application contains non-deterministic vertices, then there is no guarantee that every terminating execution produces an output that some failure-free execution could have generated.

2.5 Why You Think Your Approach is Better

A Tez-like framework generates the MapReduce paradigm to execute a complex DAG(directed acyclic graph) of tasks which allows it to meet demands for fast response and extreme throughput at petabyte scale. Since distributed data processing is the core application that Hadoop is built around, storing and analyzing large volumes and variety of data efficiently has been the cornerstone use case that has driven large scale adoption of Hadoop. Our design is to provide a framework to overcome the weaknesses of Hadoop or MapReduce and meet the movement of compute platform of Hadoop from last generation to next phase with YARN.

A Tez-like framework models data processing as a dataflow graph with vertices in the graph representing application logic and edges representing movement of data. A rich dataflow definition API allows users to express complex query logic in an intuitive manner and it is a natural fit for query plans produced by higher-level declarative applications like Hive and Pig. A tez-like framework also models the user logic running in each vertex of the dataflow graph as a composition of Input. Input and output determine the data format and how and where it is read/written. Processor holds the data transformation logic.

3 THEORETICAL BASES AND LITERATURE REVIEW

3.1 Theoretical bases and literature review.

For many resource-intensive applications, the simplest way to achieve scalable performance is to exploit data parallelism. There has been a great deal of work historically that discover and exploit parallelism in sequential programs, and some of them require the developer to explicitly expose the data dependencies of a computation. Condor was an early example of such a system in a distributed setting with shader languages developed for graphic processing units. Dryad implemented by microsoft is designed to scale from powerful multi-core single computers through small clusters of computers, to data centers with thousands of computers. The execution of Dryad handles all the difficult problems of creating a large distributed, concurrent applications.

3.2 Definition of the problem

A tez-like framework allows for efficient acquisition of resources from YARN along with extensive reuse of every component in the pipeline such that no operation is duplicated unnecessarily. These efficiencies are exposed to user logic such that users can leverage this for efficient caching and avoid work duplication.

3.3 Theoretical background of the problem

The DAG defines the dataflow of the application, and the vertices of the graph defines the operations that are to be performed on the data. The “computational vertices” are written using sequential constructs, devoid of any concurrency or mutual exclusion semantics. The

runtime parallelizes the data flow graph by distributing the computational vertices across various execution engines which can be multiple processor cores on the same computer or different physical computers connected by a network, as in a cluster.

3.4 Related research to solve the problem

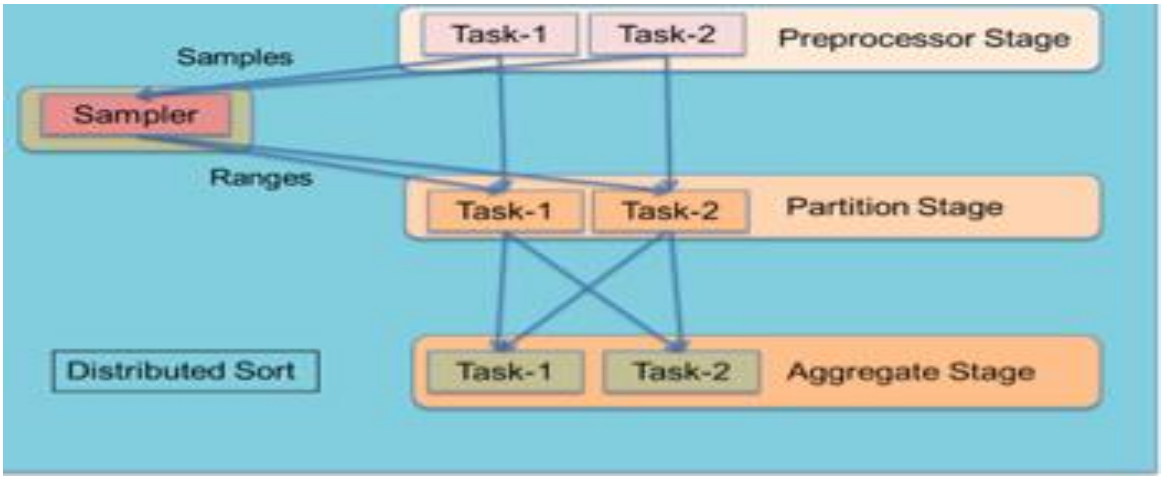
Dryad was a research project at Microsoft Research for a general purpose runtime for execution of data parallel applications. Apache Tez also generates the MapReduce paradigm to execute a complex DAG of tasks. It also represents the next logical step for Hadoop 2 and the introduction of YARN and its more general purpose resource management framework.

3.5 Advantage/disadvantage of those research

The resource acquisition in a distributed multi-tenant environment is based on cluster capacity, load and other quotas enforced by the resource management framework like YARN. The advantage of Apache Tez is that it allows to efficiently use of all available resources to run a job as fast as possible during one instance of execution and predictably over different instances of execution. The tez execution engine framework allows for efficient acquisition of resources from YARN with extensive reuse of every component in the pipeline such that no operation is duplicated unnecessarily.

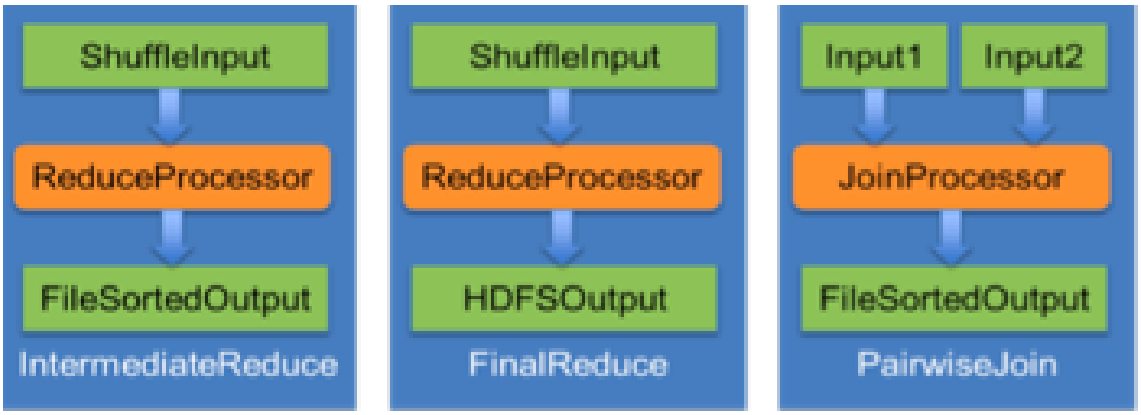
3.6 Solution

A Tez-like framework models data processing as a data flow graph with vertices in the graph representing application logic and edges representing movement of data. For example, this diagram shows how to model an ordered distributed sort using range partitioning. The Preprocessor stage sends samples to a Sampler that calculates sorted data ranges for each data partition such that the work is uniformly distributed. The ranges are sent to Partition and Aggregate stages that read their assigned ranges and perform the data scatter-gather. This data flow pipeline represents the entire computation. A Tez-like framework will expand this logical graph into a physical graph of tasks and then execute it.



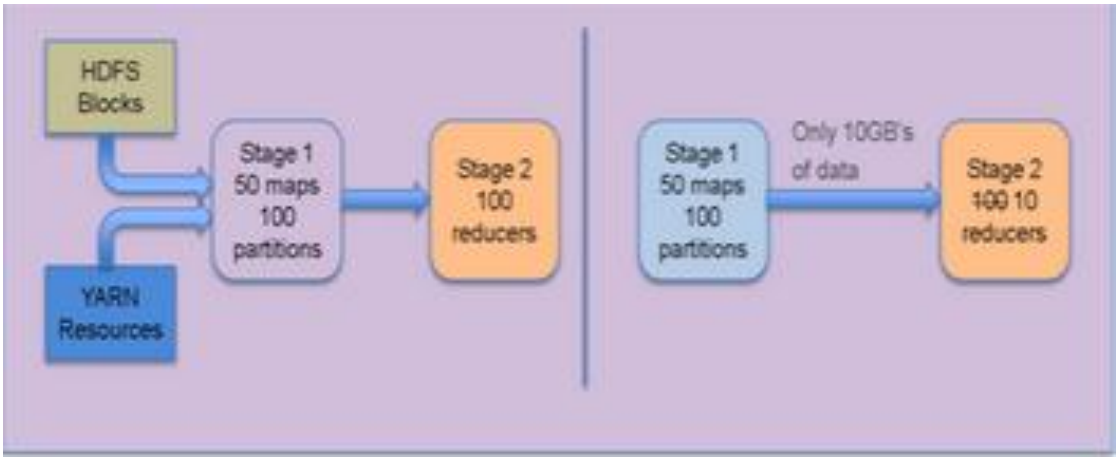
3.7 Where your solution different from others

A Tez-like framework should provide flexible Input-Processor-Output task model which let user logic running in each vertex of the data flow graph as a composition of Input, Processor and Output modules. Input and Output determine the data format and how and where it is read/written. Processor holds the data transformation logic. The framework does not impose any data format and only requires that a combination of Input, Processor and Output must be compatible with each other with respect to their formats when they are composed instantiate a vertex task. Similarly, an Input and Output pair connecting two tasks should be compatible with each other. The diagram below shows how composing different Inputs, Outputs and Processors can produce different tasks.



3.8 Advantage of our solution

Since the resource acquisition in a distributed multi-tenant environment is based on cluster capacity, load and other quotas enforced by the resource management framework like YARN. Thus resource available to the user may vary over time and over different executions of the job. It becomes critical to be able to efficiently use all available resources to run a job as fast as possible during one instance of execution and predictably over different instances of execution. The Tez-like framework allows for efficient acquisition of resources from YARN along with extensive reuse of every component in the pipeline such that no operation is duplicated unnecessarily. The diagram below show how the framework runs multiple containers within the same YARN container host and how users can leverage that to store their own objects that may be shared across tasks.



4 HYPOTHESIS (GOAL)

The apache tez: a new chapter in hadoop data processing blog post explains that " Higher-level data processing applications like Hive and Pig need an execution framework that can express their complex query logic in an efficient manner and then execute it with high performance. Apache Tez represents an alternative to the traditional MapReduce that allows for jobs to meet demands for fast response times and extreme throughput at petabyte scale. " If implemented properly, a Tez-like framework should be more efficient than traditional MapReduce program.

5 METHODOLOGY

5.1 Input Data Generation

The input of a Tez-like program can come from 3 different sources. Firstly, external data source, for example, twitter when you want to implement a twitter stream analysis using logistics regression; secondly, HDFS, this could be the generated output from another vertex in DAG; thirdly, output generated by another vertex, in this case, the output from another vertex is directly moved to consumer through a pipeline.

5.2 How to Solve the Problem

In order to understand how a Tez-like framework handlers different DAG tasks by performing operations in parallel, we need to understand the data processing API in a Tez-like framework. A tez-like framework models data processing as a data flow graph, specifically, a DAG, with the vertices in the graph representing processing of data and edges representing movement of data between the processing. All the user logic which provides analysis and modification sits in the vertices. Edges determine the movement of data means it decides which consumer is chosen to a specific input from a producer. The edges also determines how the data

is transferred and the dependency between the producer and consumer vertices. After the logical task is executed, the framework expands the logical graph into a physical graph by paralleling at the vertices. In this case, there are multiple tasks per logical vertex parallelly.

The processing starts at the root of a DAG and continues down the directed edges until it reaches the leaf vertices since there no cycle in the graph. A Tez-like framework provides fault tolerant mechanism by re-execute failed tasks whenever the input to task is lost. The framework is able to walk up the graph edges to locate a non-failed task from there to re-execute the computation.

The DAG definition API is composed of three elements: DAG, Vertex and Edge.

A DAG is an object which describes the overall job; the vertex defines the user logic and the resources & environment needed to execute the user logic. The user creates a vertex object for each step in the job and adds it to the DAG; The edge defines the movement of data from producer to consumer which connects producer to consumer.

Basically, there are three kinds of data movement. The first one is One-To-One which means data from the ith producer routes to the ith consumer; the second type is broadcast which means data from one producer task routes to all consumer tasks; the third type is scatter-gather which means producer scatter data into shards and then the shards will be available to all the consumers to gather the shards, different consumer will get different pieces of shards which depends on the implementations, for most cases, the ith shard routes to the ith consumer task.

There are two other functions in the API that needs to be added to a Tez-like framework task: scheduling and data source. Scheduling defines when a consumer task is scheduled, either sequential or concurrent. Sequential means a consumer task may be scheduled directly after a producer task finishes; while concurrent means a consumer task must be co-scheduled with the producer task. Data source determines the lifetime of a producer output. The output from a producer may be persisted (available in memory after the task finishes), persisted-reliable(reliably stored in HDFS and will always be available) or ephemeral (available only when the producer task is running).

5.3 Output Generation

The output in a DAG is generated within a vertex by the processor using the input from incoming edge and moved to the outgoing edge after it is executed. Depending on the types of the scheduling either sequential or concurrent.

5.4 Hypothesis verification

The hypothesis is verified based on the comparison of running time of an iterative application between a regular MapReduce program and a Tez-like program. Since a Tez-like framework allows data and intermediate results reuse, the running time of a Tez-like program should be much less than MapReduce.

6. IMPLEMENTATION

The goal of our implementation is to demonstrate the advantage of Tez-like framework with traditional hadoop mapreduce in interactive and iterative applications. For interactive application, we use Hive on Tez by generating tables and run data analysis query based on these tables on both hadoop and tez-like platform. for iterative applications, we will run the famous wordcount program both on hadoop and tez platform. How do we implement these two programs?

The installation of Hadoop and Tez platform and the traditional implementation of these two program will not be explained in detail in this document. The main focus of this document is on the implementations of these two program on Tez - like program, that is, how to write a custom Input/Processor/Output along with the examples of existing Input/Processor/Output task provided by the Tez-like runtime library.

The map processor or a reduce processor is a task vertex which is constituted of all the inputs from the incoming edges and outputs to the outgoing edges. And the DAG object is constructed by adding these vertices and edges. In our implementation there are two tasks or vertices.

In each of the processor class, the input, processor and output are initialized via the respective initialize methods. And configuration and context information is also made avaiable to the Input/Processor/Output via this call. The main logic of map and reduce is in run method in the respective processor. And the run method will be called with the initialized Input and Output passed in as arguments with, for example, map processor vertex input or output. After the run method, the input, processor and output will be closed as the task is considered to be complete.

The context objects is associated to provide specific information with Input/Processor/Output as a byte array. It is known that the Input and Outputs exist as a pair, that is to say, the input knows how to process DataMovementEvent generated by the corresponding Output, and how to interpret the data.

A logicalOutput are considered to have two main responsibilities: one is to deal with the actual data provided by the processor and partition it for the physical edges and serialize it; the other is to provide information to Tez on where the data is available. In our implementation, we use Writer getWriter() interface in a Processor to write result to this output.

The main responsibility f a logical input are to obtain the actual data over the physical edges and interpret the data and provide the single logical view of this data to the processor. In our implementation, we use Reader interface to expose the data to the map or reduce processor which involves interpreting the data, manipulating it, for example, decompression, etc.

To write a tez logical processor, a logical processor receives configured logical Input and logical Output. It is responsible to read source data from the input, processing it, and write the data out to the configured output. A processor is aware of which which vertex a specific input is from. Similarly, it is aware of the output vertex associated with a specific output. It would typically validate the input and output types, process the inputs based on the source vertex and

generate output for the various destination vertices.

Specifically, in our implementation, the map processor validates that it is configured with only a single input of type: MRInput since that is the only input it knows how to work with. It also validates the output to be a MROutput. A specific interface for a logical Processor is like:

```
run(Map<String, LogicalInput> inputs, Map<String, LogicalOutput> outputs)
```

This is where a processor should implement its compute logic. It receives initialized inputs and outputs along with the vertex names to which these input and outputs are connected.

The most common interfaces to be implementated by Input/Processor/Output are:

```
List<Event> initialize(Tez*Context)
```

this is where I/P/O receive their corresponding context objects. They can return a list of events.

```
handleEvents(List<Event> events)
```

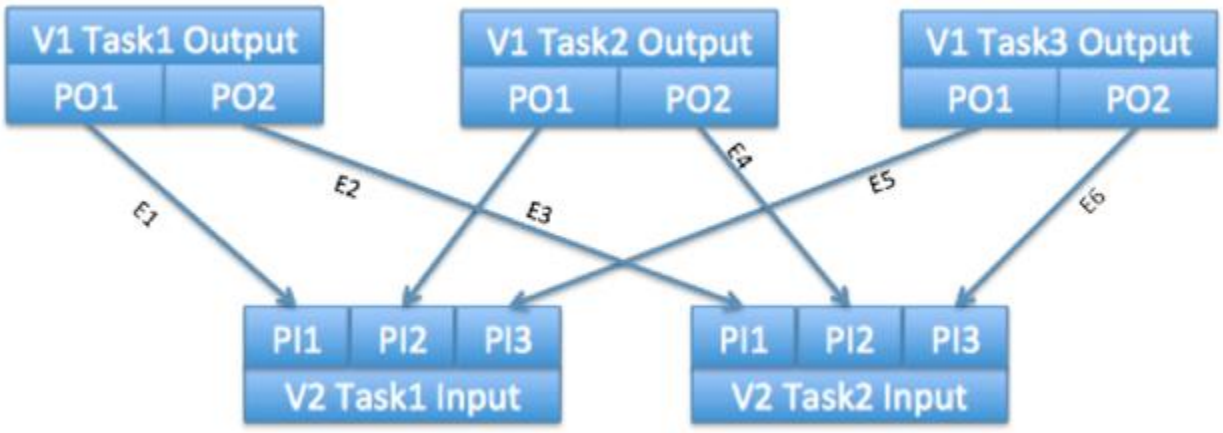
this interface is to pass the generated I/P/O

```
List<Event> close()
```

this interface is to implement cleanup and final commits.

A `DataMovementEvent` is used to communicate between Outputs and Inputs to specify location information. A byte payload is available for this, and the contents of which should be understood by communicating outputs and inputs. This byte payload could be interpreted by user-plugins running within the AM to modify the DAG.

`DataMovementEvent` are typically generated per physical edge between the output and input. The event generator needs to set the sourceIndex on the event being generated; and this matches the physical output/input that generated the event. Below is an example to illustrate this event:



in this case, the input has 3 tasks, and the output has 2 tasks. Each input generates 1 partition for the downstream tasks, and each downstream task consumes the same partition from each of the upstream tasks.

7. DATA ANALYSIS AND DISCUSSION

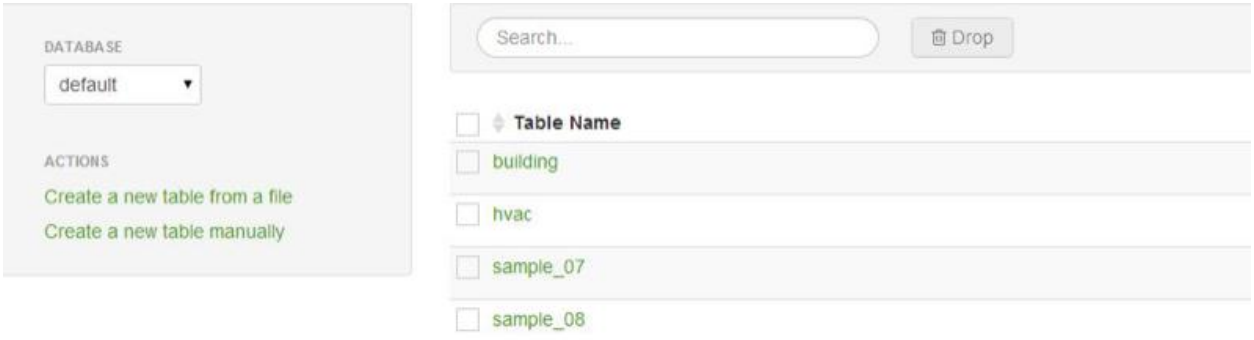
Our data analysis is focused on demonstrate the advantages and improvements of Hive on Tez by creating tables on Hadoop Tez and running queries using Hadoop Tez compared with result of mapreduce tez. Since hadoop tez will reuse immediate results and storing results in memory, thus the performance improvements of Hive on Tez can be achieved.

first, we create two tables into HDFS.

Table preview

	Column name	Column name	Column name	Column name
	date	time	targettemp	actualtemp
	Column type	Column type	Column type	Column type
	string	string	bigint	bigint
Row #1	6/1/13	0:00:01	66	58
Row #2	6/2/13	1:00:01	69	68
Row #3	6/3/13	2:00:01	70	73
Row #4	6/4/13	3:00:01	67	63
Row #5	6/5/13	4:00:01	68	74
Row #6	6/6/13	5:00:01	67	56
Row #7	6/7/13	6:00:01	70	58
Row #8	6/8/13	7:00:01	70	73
Row #9	6/9/13	8:00:01	66	69

after creating two tables “Hvac” and “building”

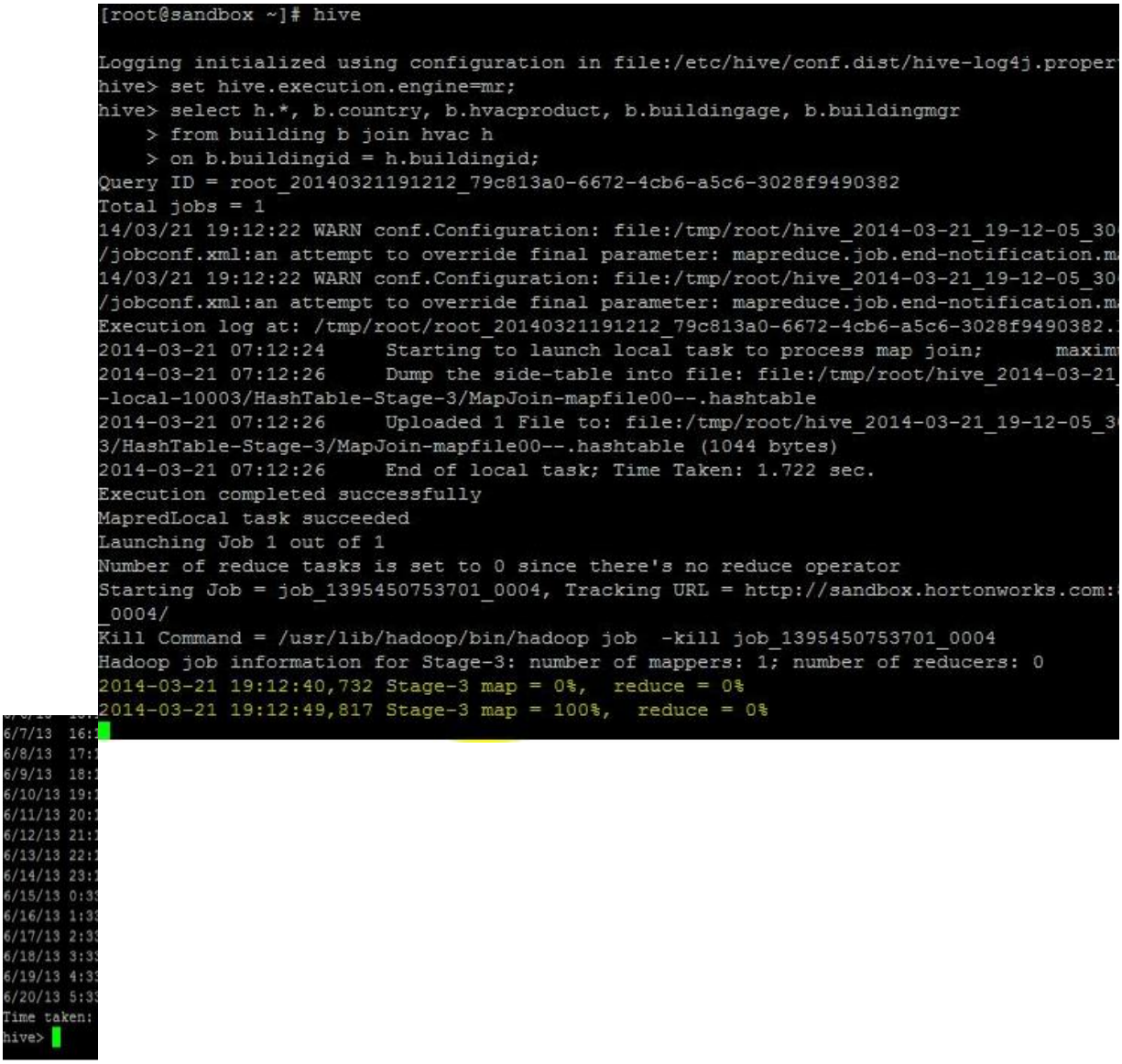


Then we first run Hive without Tez:

```
set hive.execution.engine=mr;
```

```
select h.*, b.country, b.hvacproduct, b.buildingage, b.buildingmgr
from building b join hvac h
on b.buildingid = h.buildingid;
```

mapreduce Hive is running the query we wrote:



The result shows that is takes mapreduce Hive 47 seconds to execute the query

Now we use Hive on Tez to execute the same query. Since Hive on Tez represent the query as a DAG and write intermediate results into memory instead of hard disk which incurs substantial overhead involved with synchronization and I/O. The speed performanced should be improved:

```
set hive.execution.engine=tez;
```

```
select h.*, b.country, b.hvacproduct, b.buildingage, b.buildingmgr
from building b join hvac h
on b.buildingid = h.buildingid;
```

```
[root@sandbox ~]# hive

Logging initialized using configuration in file:/etc/hive/conf.dist/hive-log4j.properties
hive> set hive.execution.engine=tez;
hive> select h.*, b.country, b.hvacproduct, b.buildingage, b.buildingmgr
    > from building b join hvac h
    > on b.buildingid = h.buildingid;
Query ID = root_20140321192020_e84dc598-43d0-408b-bb79-6be20698e41d
Total jobs = 1
Launching Job 1 out of 1


Status: Running (application id: application_1395450753701_0005)

Map 1: -/-      Map 2: -/-
Map 1: 0/1      Map 2: 0/1
Map 1: 0/1      Map 2: 0/1
```

```
6/4/13  13:13:20        66      66      2       24      11      Belgium AC1000  14      M11
6/5/13  14:13:20        67      68      4       22      20      Argentina       ACMAX22 19      M20
6/6/13  15:13:20        67      79      5       8       3       Brazil  JDNS77  28      M3
6/7/13  16:13:20        67      62      9       14      2       France  FN39TG  27      M2
6/8/13  17:13:20        65      75      13      13      10      China   ACMAX22 23      M10
6/9/13  18:13:20        67      65      4       17      13      Saudi Arabia    JDNS77  25      M13
6/10/13 19:13:20        70      55      20      12      4       Finland GG1919  17      M4
6/11/13 20:13:20        65      68      15      2       14      Germany GG1919  17      M14
6/12/13 21:13:20        68      74      1       30      13      Saudi Arabia    JDNS77  25      M13
6/13/13 22:13:20        66      61      4       15      20      Argentina       ACMAX22 19      M20
6/14/13 23:13:20        67      55      3       14      14      Germany GG1919  17      M14
6/15/13 0:33:07 70      60      2       9       19      Canada  GG1919  14      M19
6/16/13 1:33:07 66      58      17      18      20      Argentina       ACMAX22 19      M20
6/17/13 2:33:07 68      72      17      27      12      Finland FN39TG  26      M12
6/18/13 3:33:07 68      69      10      4       3       Brazil  JDNS77  28      M3
6/19/13 4:33:07 65      63      7       23      20      Argentina       ACMAX22 19      M20
6/20/13 5:33:07 66      66      9       21      3       Brazil  JDNS77  28      M3
Time taken: 27.437 seconds, Fetched: 8000 row(s)
```

In this graph above, it shows that it takes 27 seconds to run the same query on Hive on Tez. Compared with the time on Hive without Tez which is 47 seconds, the speed has improved a lot which proved our assumption.

To show the improvement further, we run the same query from last step,

```
select a.buildingid, b.buildingmgr, max(a.targettemp-a.actualtemp)
from hvac a join building b
on a.buildingid = b.buildingid
group by a.buildingid, b.buildingmgr;
```

the result shows that:

```
6/12/13 21:13:20        68      74      1       30      13      Saudi Arabia    J
DNS77   25      M13
6/13/13 22:13:20        66      61      4       15      20      Argentina       A
CMAX22  19      M20
6/14/13 23:13:20        67      55      3       14      14      Germany GG1919 1
7       M14
6/15/13 0:33:07 70      60      2       9       19      Canada  GG1919  14      M
19
6/16/13 1:33:07 66      58      17      18      20      Argentina       ACMAX221
9       M20
6/17/13 2:33:07 68      72      17      27      12      Finland FN39TG  26      M
12
6/18/13 3:33:07 68      69      10      4       3       Brazil  JDNS77  28      M
3
6/19/13 4:33:07 65      63      7       23      20      Argentina       ACMAX221
9       M20
6/20/13 5:33:07 66      66      9       21      3       Brazil  JDNS77  28      M
3
Time taken: 14.862 seconds, Fetched: 8000 row(s)
hive>
```

Now it shows that it only takes 14 seconds to execute the same query which is also a big improvement.

8 CONCLUSION AND RECOMMENDATIONS

A Tez-like framework provides a performance improved framework for data processing across a large set of methods, from batch through interactive to real time. For Hive on Tez, the tables in Hive are similar to tables in a traditional relational database. It supports overwriting or appending data but not updates and deletes. To make data processing on Hive more efficient, data in Tez are serialized in a particular database, and each table has a corresponding Hadoop Distributed File System(HDFS) directory. Each table can be sub-divided into partitions that determine how data is distributed within sub-directories of the table directory. Data within partitions are further divided into buckets. Compared with relational database, data units in Tez are more granular units.

We have proved and showed that the speed performance of Hive on Tez is much more efficient than the performance of mapreduce Hive. And the response times are faster than other types of queries on the same types of huge datasets.

For regular applications, Tez represents the implementation as a DAG object. Each object is a task composed of different vertices or processors. The main logic of the implementation is in the respective vertex or processor. Since Tez store the immediate results into memory instead of hard disk, each processor or vertex in the DAG need to specify the input and output in order to communicate between the vertices. For applications with extensive I/O overhead or applications involves a lot of immediate result reuse, Tez outperforms traditional mapreduce framework in a substantial way. Our implementation of Ordered Word Count proves this efficiency.

9 BIBLIOGRAPHY

1) Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, Dennis Fetterly, Microsoft Research. 2007.
2) Writing a Tez InputProcessor/Output Siddharth Seth, October, 2013.