

Project number:	317871
Project acronym:	BIOBANKCLOUD

<p>WORK PACKAGE 2 :</p> <p>SCALABLE STORAGE</p>

Work Package Leader Name and Organisation:

Jim Dowling, KTH – Royal College of Technology (KTH)

E-mail: jdowling@kth.se

PROJECT DELIVERABLE

D2.1: Highly Available HDFS

Deliverable Due date (and month since project start): 2013-11-30, m12

Document history

Version	Date	Changes	By	Reviewed
0.1	2013-11-20	First version	Salman Niazi Kamal Hakimzadeh Alberto Lorente Mahmoud Ismail Jim Dowling	Ali Gholami
0.2	2013-11-29	Incorporated comments from Ali G.	Jim Dowling Salman Niazi	

BiobankCloud D2.1

317871

Executive Summary

This deliverable is a software deliverable of a highly available implementation of the Hadoop Filesystem (HDFS). In this document, there a description of the system's architecture, Platform-as-a-Service support for HDFS, and a userguide for the software.

Our new version of HDFS contributes a new high availability model for HDFS' metadata, based on storing the metadata in MySQL Cluster. MySQL Cluster is a distributed, in-memory, highly available, high performance database. The main contribution of our new architecture is a strengthening of the replication model from the Apache's HDFS, which is based on eventually consistent primary-secondary replication. In our model, a set of stateless NameNodes read and write metadata from shared transactional memory in MySQL Cluster. This enables us to simplify some of HDFS' internal protocols as well as enabling a larger number of NameNodes (as opposed to only a primary and secondary NameNode in HDFS). Our implementation also maintains the consistency semantics of HDFS, and we validate this by ensuring that 300+ unit tests pass for HDFS.

We also describe the Platform-as-a-Service (PaaS) support we provide for our HDFS implementation. HDFS is the storage layer of the Hadoop ecosystem, and we automate the installation of both HDFS and Apache YARN, enabling our platform to be installed by unsophisticated users by just clicking options and entering user credentials from our portal website. We support automated deployment for the following platforms: Amazon Web Services (AWS), OpenStack or a standard cluster of (bare-metal) hosts. We also provide a Dashboard web application to administer and monitor deployed Hadoop clusters.

This document also includes a userguide for installing and managing our platform, that we call Hop (Hadoop Open Platform-as-a-service). It is *open*, as it can be extended to support any existing cloud platform, even though we currently only support AWS and OpenStack. The code is available for download now, although it is still very much beta and under heavy development.

Table of Contents

1. Hadoop Open PaaS	1
Highly Available Hadoop Filesystem (HDFS)	1
Leader Election	3
Ensuring Correctness of Hop HDFS	7
Snapshot layer	10
Performance of Hop HDFS	12
Hop: Platform-as-a-Service support for HDFS	13
2. Quickstart with Vagrant	18
Pre-requisites	18
Launching Vagrant	18
3. Hop Web Portal	19
Prerequisites	19
Installing the Hop Dashboard	19
Amazon EC2	20
OpenStack	21
Baremetal Hosts	22
4. Hop Dashboard	24
Graphs	24
Backup/Restore	25
Setup Credentials	25
Cluster Management	25
Cluster Deployment Progress	26
Monitoring	27
Hosts	27
Alerts	28
Clusters	28
5. Defining a Cluster	33
Cluster Wizard	36
6. Configuring HDFS	41
HDFS Configuration Parameters not used	41
Additional HDFS Configuration Parameters	41
7. Conclusions	43

List of Figures

1.1. Hadoop v2	1
1.2. HDFS v2 NameNode Primary/Secondary Replication Model	2
1.3. Hop HDFS	3
1.4. Leader Table in NDB	4
1.5. Leader Table in NDB	5
1.6. Entity-Relation Diagram for NameNode state in Hop-HDFS	8
1.7. HDFS Filesystem Operation Sequence Diagram	9
1.8. Locking in conflicting order can lead to Deadlock	10
1.9. Lock upgrade can lead to Deadlock	10
1.10. Reduction in the number of DB roundtrips by snapshotting	13
1.11. Hop architecture	13
1.12. Deploying a Hop cluster from our Portal Website	14
1.13. Hop PaaS API	15
3.1. Hop Portal Website	19
3.2. AWS Credentials	20
3.3. OpenStack Credentials	21
3.4. Baremetal Credentials	22
4.1. Graph Editor	24
4.2. Graph Selection Detail	25
4.3. Manage Cluster	25
4.4. YAML Editor	26
4.5. Hosts	27
4.6. Hosts Services	27
4.7. Host Graphs	28
4.8. Alerts	28
4.9. Clusters	28
4.10. Detailed Cluster View	29
4.11. YARN Metrics	29
4.12. Resource Manager Metrics	30
4.13. Node Manager Metrics	30
4.14. Resource Manager UI	30
4.15. Node Manager UI	31
4.16. MySQL Cluster Monitor	31
4.17. MySQL Console	32
4.18. Hdfs Console	32
5.1. Common Cluster Options:	37
5.2. Bare Metal Common Cluster Options:	37
5.3. Cluster Provider Options:	38
5.4. Cluster Group:	39
5.5. Bare Metal Groups:	39
5.6. Confirmation	40

List of Examples

1.1. Leader Election Algorithm at NameNodes	6
1.2. Detecting the Leader at DataNodes	7
1.3. Snapshotting at NameNodes	11
1.4. Snapshotting at NameNodes using Two Transactions	12
1.5. Example Cluster Definition in Hop	15
5.1. Complete Example Cluster Definition in Hop	34
5.2. OpenStack provider	35
5.3. Bare-metal cluster	35

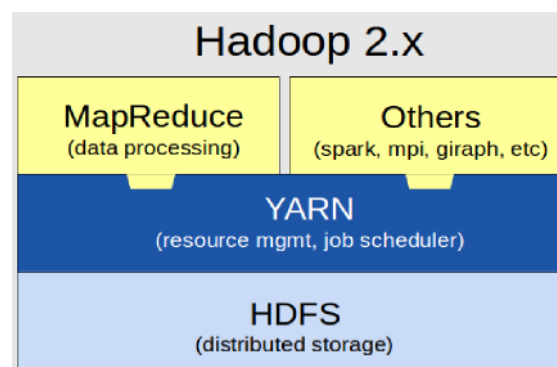
Chapter 1. Hadoop Open PaaS

In this chapter, we introduce our implementation of the the Hadoop Filesystem, where we introduce a new model for the NameNode that enables clusters with larger, more configurable metadata than is currently found in Apache's HDFS. We also introduce a Platform-as-a-Service (PaaS) model for our HDFS implementation, showing how we automate the deployment of a cluster on cloud platforms, such as Amazon Web Services (AWS) and OpenStack, as well as standard clusters.

Highly Available Hadoop Filesystem (HDFS)

Due to the rapid growth of data in recent years, distributed file systems have gained widespread adoption. The new breed of distributed file systems reliably store petabytes of data, and also provide rich abstractions for massively parallel data analytics [5,7]. The Hadoop Distributed File System (HDFS) [3] is a distributed, fault-tolerant file system designed to run on low-cost commodity hardware that scales to store petabytes of data, and is the file storage component of the Hadoop platform [3,4]. HDFS provides the storage layer for MapReduce, Hive, HBase, Spark and all other YARN applications, see Figure 1.1, “Hadoop v2”.

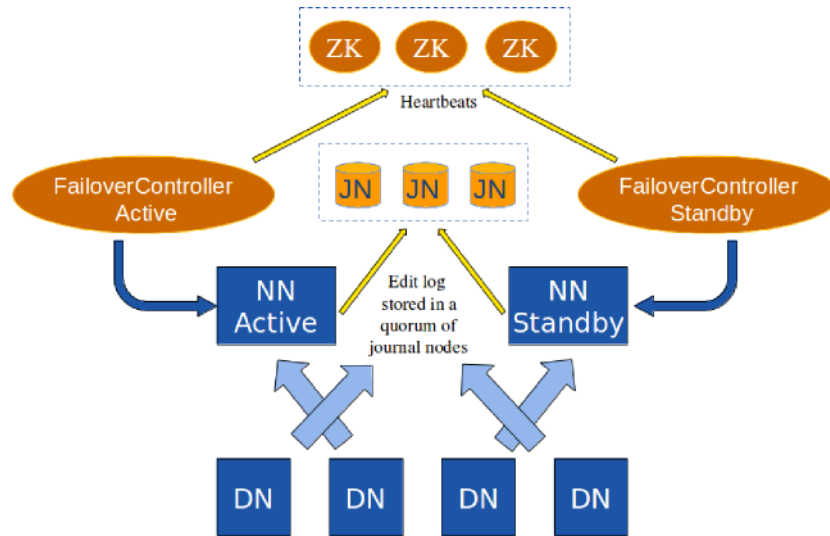
Figure 1.1. Hadoop v2



In 2013, HDFS v2 introduced a highly available metadata architecture [1], where the entire filesystem's metadata is stored in memory on a single node [13], but changes to the metadata (*edit log entries*) are now replicated and persisted to a set of (at least three) Journal Nodes using a quorum-based replication algorithm, see Figure 1.2, “HDFS v2 NameNode Primary/Secondary Replication Model”. In HDFS v2, a Primary and Secondary NameNode can be configured, where the Primary NameNode is responsible for managing the metadata, and the Secondary NameNode keeps an eventually consistent copy of the metadata. The Secondary NameNode is kept in sync with the Primary by two mechanisms: firstly, by asynchronously applying all edit log entries that have been committed at the Journal Nodes, and secondly, receiving the same set of heartbeats from Data Nodes that are received by the Primary. This Primary/Secondary replication model is also known as an Active/Standby or Master/Slave replication model, and was popularized by databases in the 1990s. HDFS' implementation of this eventually consistent replication model is more limited than in the traditional relational database world, as all read and write requests are sent to the Primary. In typical Master/Slave configurations, writes are sent to the master, while reads are load-balanced across slaves. The reason all write requests are sent to the Primary to ensure a single consistent copy of the metadata. Read requests are also sent to the Primary, as reads at the Secondary could result in operations being executed on stale metadata. This is a bigger problem for a filesystem, such as HDFS, than it would be for a Web 2.0 social media application with non-critical data, using a Master/Slave database setup. Thus, reads are only sent to the Primary. If the Primary fails, however, the Secondary needs to take over as Primary. Before it can take over, it first has to catch up with the set of edit log entries applied to Primary before it failed. The period of time before all outstanding edit log entries are applied at the Secondary before it can take over may be up to tens of seconds, depending on the current load of the system and the hardware and software

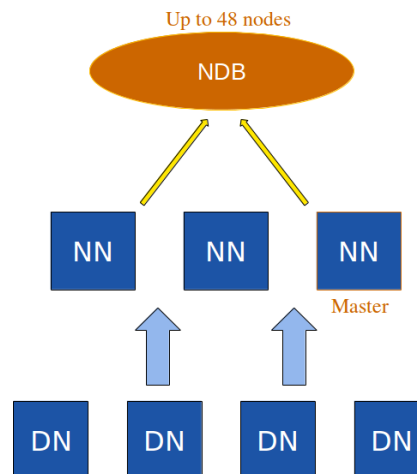
setup. Another limitation of the Primary/Secondary model, is that client and Data Nodes from HDFS need to have a consistent view of who the current Primary NameNode is. They do this by asking a Zookeeper coordination service that needs to run on at least 3 nodes to provide a fault tolerant reliable service. Finally, the concurrency model supported by HDFS v2 is still multiple-readers, single-writer.

Figure 1.2. HDFS v2 NameNode Primary/Secondary Replication Model



Hop HDFS

In contrast, our implementation of HDFS, called *Hop HDFS*, replaces the Primary-Secondary metadata model with shared, transactional memory, implemented using a distributed, in-memory, shared-nothing database, MySQL Cluster (NDB) [11] see Figure 1.3, “Hop HDFS”. MySQL Cluster is a real-time, ACID-complaint transactional database, with no single point of failure, that has predictable, milisecond response times with the ability to service millions of operations per second. By leveraging MySQL Cluster to store HDFS' metadata, the size of HDFS' metadata is no longer limited to the amount of memory that can be managed on the heap of JVM on a single node. Our solution involves storing the metadata in a replicated, distributed, in-memory database that can scale up to several tens of nodes, all while maintaining the consistency semantics of HDFS. We maintain the consistency of the metadata, while providing high performance, all within a multiple-writer, multiple-reader concurrency model. Multiple concurrent writers are now supported for the filesystem as a whole, but single-writer concurrency is enforced at the inode level. Our solution guarantees freedom from deadlock and progress by logically organizing inodes (and their constituent blocks and replicas) into a hierarchy and having transactions defining on a global order for transactions acquiring both explicit locks and implicit locks on subtrees in the hierarchy; this solution is motivated by [9] and [2]. The use of a database, however, also has its drawbacks [8,12]. As the data now resides on remote hosts on the network, an excessive number of roundtrips to the database harms system scalability and increases per-operation latencies. We ameliorate these problems by introducing a snapshotting mechanism for transactions, where, at the beginning of a transaction, all the resources it needs are aquired in the defined global order, while simulatenously taking row-level locks for those resources. On transaction commit or abort, the resources are freed. This solution enables NameNodes to perform operations on a local copy (or snapshot) of the database state until such time as the transaction is completed, thus reducing the number the number of roundtrips required to the database, see Figure 1.10, “Reduction in the number of DB roundtrips by snapshotting” .

Figure 1.3. Hop HDFS

Leader Election

In HDFS, there are a number of background tasks that are problematic if multiple NameNodes attempt to perform them concurrently. Examples of such tasks include:

1. replication monitoring,
2. lease management,
3. block token generation,
4. and the decommissioning of datanodes.

Without any coordination between NameNodes, and since all NameNodes have identical behaviour, we would have problems with many of these background tasks. For example, if a block becomes under-replicated, several NameNodes could independently identify this under-replicated state, and each would select a DataNode to replicate that block to. This would cause multiple re-replications of the block, leading it to enter an over-replicated state, upon which, multiple NameNodes could recognize this over-replicated state and attempt to remove replicas, possibly leading to an under-replicated state - back where we started. We solve this coordination problem, by implementing a Leader Election Algorithm, where only the leader NameNode is assigned the task of performing the above background tasks. Our leader election algorithm uses the shared, transactional memory abstraction provided by MySQL Cluster to coordinate the election process.

Definition: Correct NameNode A correct NameNode is defined as an alive process that is able to write to NDB in a bounded time interval.

Definition: Leader NameNode A leader NameNode is a NameNode from the set of NameNodes that is correct and is responsible for listening to DataNode heartbeats and assigning various tasks to them as well as responsible for managing background tasks.

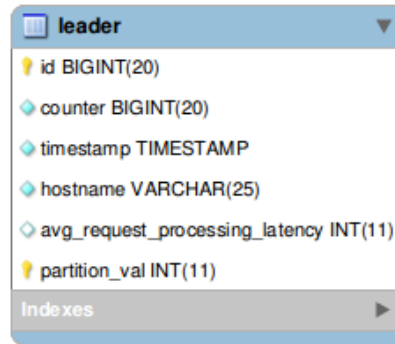
In a deployed system, the bounded time interval during which it is expected that a NameNode can write to NDB is typically set to around 1 second. The properties that our leader election algorithm guarantee are:

1. *Completeness*: after a bounded time interval, all correct NameNodes will detect every NameNode that has crashed.
2. *Agreement*: after a bounded time interval, all correct NameNodes will recognize one among them as the leader. All will agree to the same NameNode being the leader.
3. *Stability*: If one correct NameNode is the leader, all previous leaders have crashed

Leader Election Algorithm

The leader election algorithm runs continuously at the NameNodes in rounds. Each NameNode is assigned an (integer) id. At any point in time, the correct NameNode with the lowest id is elected as the leader. The central focus of the algorithm is to detect failures and elect a new leader using heartbeats. We implement heartbeats as counters in a table in NDB. The algorithm is run at NameNodes in rounds (every n th second), and it is expected that each NameNode will send only one heartbeat per round, although our algorithm tolerates minor deviations, as explained later. NameNodes can discover who the leader is by reading the ids of correct NameNodes from a table in NDB. NDB's shared transactional memory allows all NameNodes to have a uniform view of the correct NameNodes in the system. We implemented our heartbeat model using the schema shown in Figure 1.4, "Leader Table in NDB". Some sample records are also shown to show an example run of the algorithm.

Figure 1.4. Leader Table in NDB



The image shows a database schema for a table named 'leader'. The columns are: id BIGINT(20), counter BIGINT(20), timestamp TIMESTAMP, hostname VARCHAR(25), avg_request_processing_latency INT(11), and partition_val INT(11). There is an 'Indexes' section at the bottom with a right-pointing arrow.

Column	Data Type
id	BIGINT(20)
counter	BIGINT(20)
timestamp	TIMESTAMP
hostname	VARCHAR(25)
avg_request_processing_latency	INT(11)
partition_val	INT(11)

Indexes

Once a NameNode starts, it starts a rounds timer which periodically triggers causing it to send a heartbeat to the NDB to indicate that it is currently active and running. The heartbeat is implemented by a NameNode first reading the highest counter value in a LEADER table and then updating its counter value in the LEADER table to be one higher than the current highest counter value. A heartbeat, in effect, increments the highest counter value. Heartbeats should be loosely synchronized by configuring hosts to use Network Time Protocol, and setting equivalent timeouts for triggering heartbeats at nodes. To ensure serialized updates to counter values, when each NameNode reads the highest counter value, it acquires a table-level write lock on the LEADER table and only releases this lock after it has updated its own counter value. All of these operations happen within a transaction, so that the lock is released in the event of a NameNode failure while updating the LEADER table.

Case-I: Failure-free scenario

In the case of no failures, after each round when all NameNodes have successfully sent heartbeats, the highest value of the counter should have increased by the number of correct NameNodes, and the value of each NameNode's counter should have increased by approximately the number of correct NameNodes. We say approximately, as clock skew, network delays, and congestion may cause updates to arrive at varying times within a round. Assuming clock skews are not as large as the heartbeat interval and no congestion, all NameNodes should succeed in sending one heartbeat at least every 2 rounds. The above figure shows an example of counter values for 3 NameNodes with ids 1, 2 and 3 and their corresponding counter values as 23, 24 and 25 respectively. For brevity, we designate a NameNode with an id of value x as NN_x . From this example we see that NN_1 has the lowest id and is therefore elected as the current leader in the system.

Case-II: Leader crash scenario

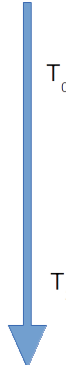
For numerous reasons, a NameNode may fail in sending a heartbeat to NDB a round, and, thus, fail to update its counter value in the LEADER table. The counter value for the NameNodes remains the same. NameNodes would experience an irregular sequence of counter values in each of these rows. For example, let's say that NN_1 has crashed and the current counter value is now 6. This would allow NN_2 and NN_3 to progress in updating the counter with values 9 and 10 while counter value for NN_1 is still at value 6. In Figure 1.5, "Leader Table in NDB", we can see a snapshot of the view on the LEADER table at this round (T_0). The figure shows non-consecutive counter values 6, 9 and 10 when we would

expect something like 8, 9 and 10. As all NNs have a consistent view of these counter values, so NN2 and NN3 can now agree that NN1 has crashed, and NN2 will become leader.

Figure 1.5. Leader Table in NDB

id	counter	timestamp	hostname
1	6	2012/10/19 22:01:10	cloud1
2	7	2012/10/19 22:01:10	cloud2
3	8	2012/10/19 22:01:11	cloud3

id	counter	timestamp	hostname
1	6	2012/10/19 22:01:10	cloud1
2	10	2012/10/19 22:01:13	cloud2
3	11	2012/10/19 22:01:13	cloud3



Determining the leader

The next step is to detect an irregular sequence of counter values and to decide if a new leader is to be elected. The basic idea to determine the leader is to determine which NameNode have counter values close enough to the current highest counter value to be considered 'correct'. In the example given above, the highest counter value at round T2 (bottom table) is 11. As we have 3 NameNodes in the system, we expect the counter values to be in the range [9-11], although allowing for out-of-order heartbeats, the counter values could be in the range [7-10]. The upper-bound on out-of-order updates to counters is, by default, 2 rounds. That means that a NameNode that is more than 2 rounds behind in updating its counter value is no longer considered to be correct. In the lower table, we can see that NN1 is now no longer considered correct, as both NN2 and NN3 have succeeded in updating their counters 3 times before NN1 has updated its counter. Therefore, NN2 now becomes the new leader, as its counter value is within the range of 2 rounds from the max counter value, and it has the lowest id of the remaining correct NNs. Both NN2 and NN3 independently reach a decision about the new leader, NN2, the next time they try to update their own counter.

Ensuring single leader dominance

Once a NameNode determines that it is the leader, the first thing it does is to ensure that there are no other leaders in the system. This means that all previously elected leaders (or NameNodes with ids lower than its own) have crashed. To ensure this, it enforces this rule by removing all records from the [LEADER] table for NNs with a lower id than its own.

Example 1.1. Leader Election Algorithm at NameNodes

```
function updateCounter()
    retrieve ([LEADER], maxCounter)
    increment maxCounter
// The entry for this NN may not exist if it crashed
// and was removed by another leader
    if (!exists([LEADER], id) then
        id = retrieve([LEADER], max(id)) + 1
    end if
    store([LEADER], id, maxCounter)

// The function that determines the current leader
function select()
    SELECT id FROM [LEADER]
    WHERE counter (max(counter) - count(id)) // returns all correct NNs
    LIMIT 1 /*selects lowest id*/
    return id

// The function that returns the list of correct NNs
function selectAll()
    SELECT id FROM [LEADER]
    WHERE counter > (max(counter) - count(id)) // returns all correct NNs
    ORDER BY id                                // the leader has the lowest id
    return list(id)

upon event <init> do
    leader = NIL
    updateCounter()
    leader = select()

// After every interval, the NameNode updates the counter
upon event <check>
    updateCounter()
    leader = select()
    // If this NN is elected the leader, remove previous leaders
    if (leader == id) then
        remove([LEADER],[ids < leader])
    end if

upon event < timeout>
    // Kill NN on timeouts, so DNS can connect to the next leader NN
    shutdown()

// Return the list of correct NNs in order of lowest ids
upon event <heartbeats>
    return selectAll()
```

Example 1.2. Detecting the Leader at DataNodes

```
// The NN leader id is passed in upon initialization of DN
upon event init<list(id)>
    nnlist = list(id)
    // The NN with the lowest id is the leader
    leader = min(nnlist)

// Update the list of NNs on every heartbeat response from Leader NN
upon event <heartbeat-response, list(id)>
    nnlist = list(id)
    leader = min(nnlist)

// On timeout from leader NN
upon event <timeout>
    // remove current leader from nnlist
    remove(nnlist, leader)
    // elect a new leader
    leader = min(nnlist)
```

All DataNodes are kept up-to-date with the current view of NameNodes in the system. This is done via requesting the NameNodes for the current list of NameNodes. This is done via a simple RPC call. The DataNodes get the list of NameNodes and assume the NameNode with the lowest id is the leader. When NN1 had crashed, the DNs keep retrying for some amount of time and if they are not successful at making contact with the NN it will remove it from the list and select the next NN who could potentially be the leader. This would achieve [Property#2]. There can be two possibilities where (a) the DataNode contacts the next NN and if it is actually the leader then the process flows normally. (b) The DataNode contacts the next NN in the list but who may not be the leader (because it is possible that it has also crashed or a new NN has joined with a lower id). This NN would then either provide the updated list of NameNodes, including the new leader, or not respond due to failure. The DataNode can continue querying all NameNodes until either it is returned a list of correct nodes (including the id of the leader) or all NameNodes have failed. If there is a correct NameNode, eventually all correct DNs would recognize a correct NN as the leader thereby fulfilling [Property#1].

Case-II: Scenario when current leader thinks it is alive but cannot connect to NDB If NN1 is active and running, but it cannot update the LEADER table in NDB, then such a NameNode is not considered correct as per definition. In this scenario, all DataNodes would always think that NN1 is the current leader as it can make contact with that NN. But since the NN cannot update NDB, it has to kill itself and shutdown (or restart) hoping that some other NameNode would eventually be elected the leader. When NN1 is shutdown, DataNodes will keep retrying for some amount of time after some point where they will switch to the next NameNode and try to determine the next NN leader.

Ensuring Correctness of Hop HDFS

One of the main challenges in migrating HDFS' metadata from the heap of a single JVM to a relational database was ensuring the correctness of our approach. From a practical perspective, we ensure correctness by ensuring that we have (almost) no failing unit tests from the extensive suite of over 300 unit tests. From a theoretical perspective, our solution provides for support for multiple concurrent writers, so we need to show that our solution is free of both deadlock and livelock. Our first challenge was to migrate the state of HDFS from highly optimized data structures in the NameNode to tables in NDB. We migrated a total of 13 datastructures from HDFS' NameNode to tables in NDB, see Figure 1.6, "Entity-Relation Diagram for NameNode state in Hop-HDFS".

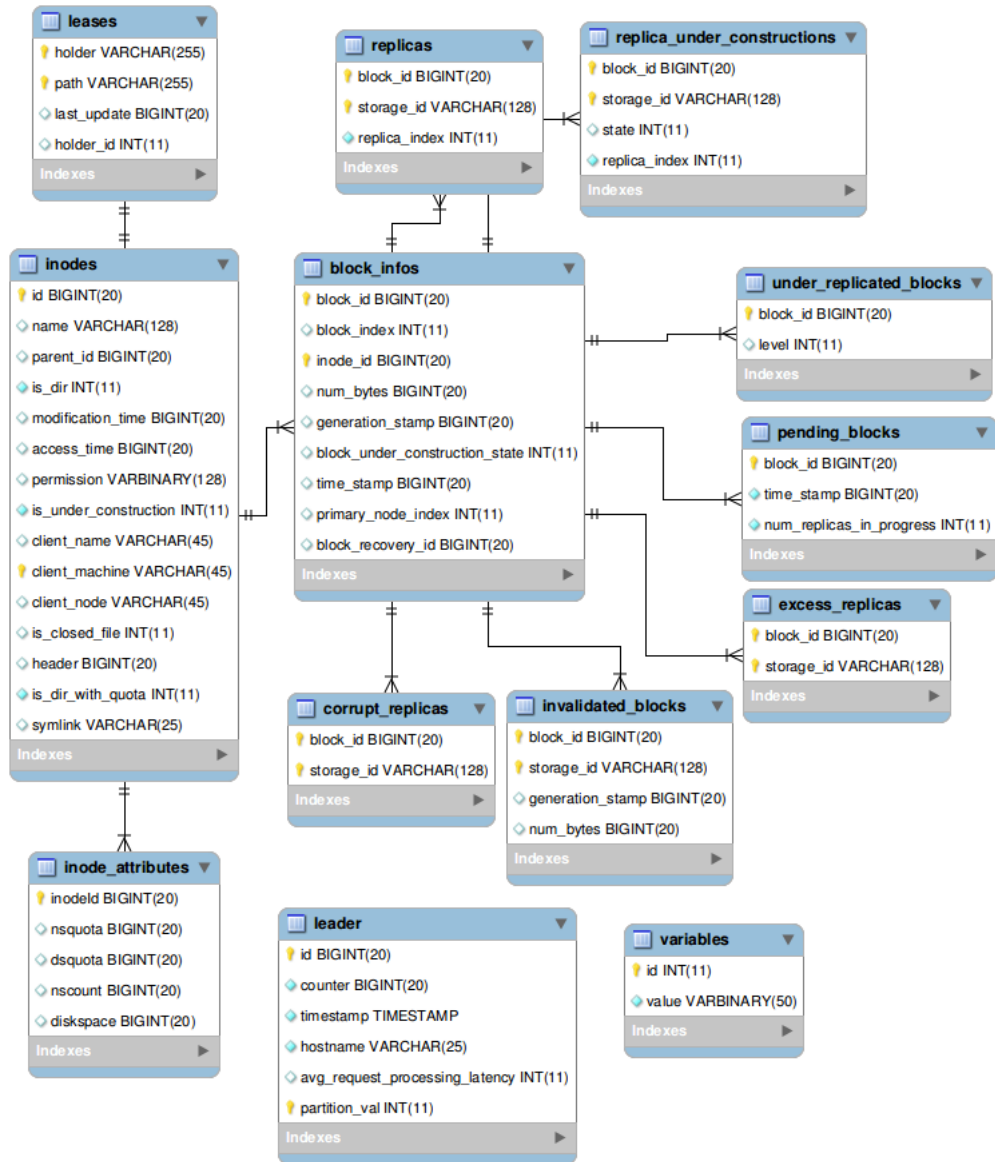
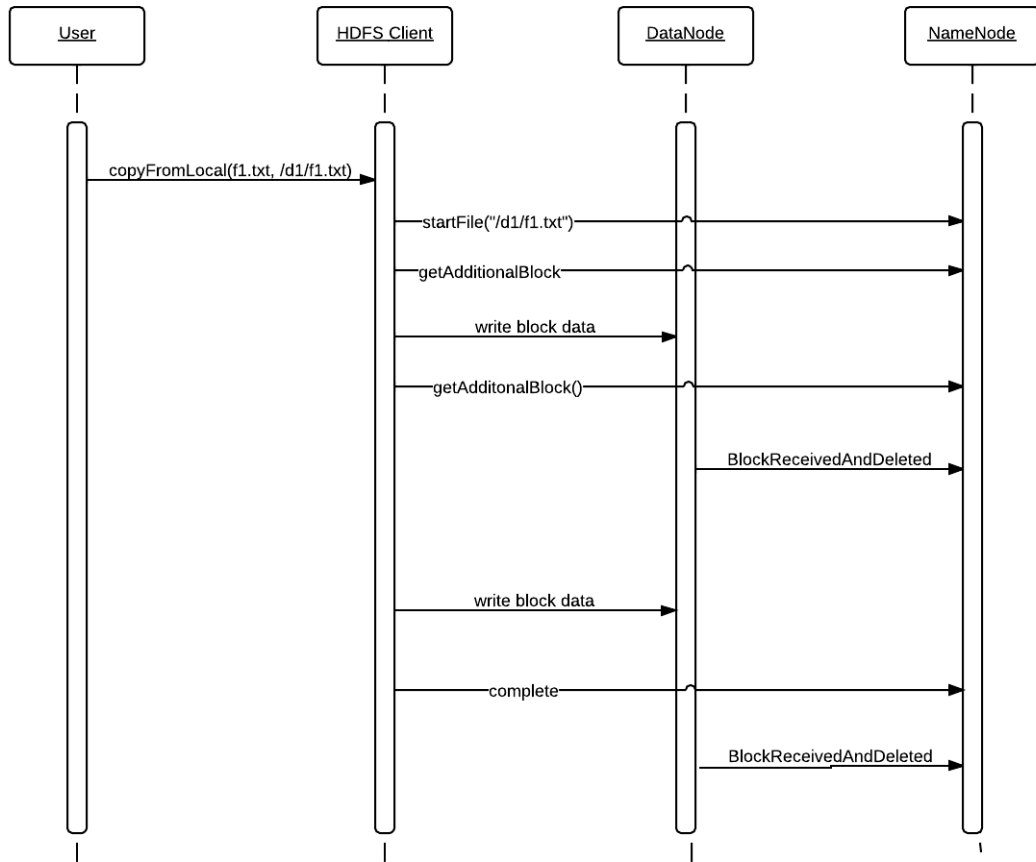
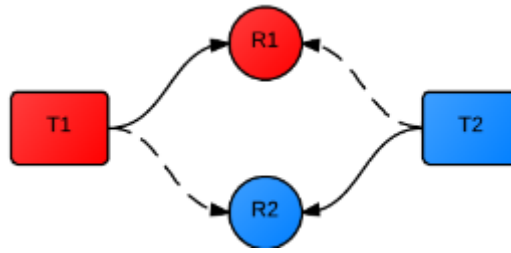
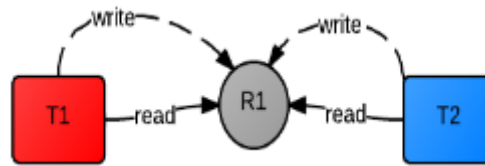
Figure 1.6. Entity-Relation Diagram for NameNode state in Hop-HDFS

Figure 1.7. HDFS Filesystem Operation Sequence Diagram*Transactions and Isolation Levels*

We implemented each HDFS operation as a single transaction, where we begin the transaction, read and write the necessary meta-data from NDB, and then either commit, or in the case of failure, abort the transaction and then possibly retry the transaction. An example of a HDFS filesystem operation that copies a file from the local filesystem into HDFS is illustrated in Figure 1.7, “HDFS Filesystem Operation Sequence Diagram”. In this example, we can see that the `copyFromLocal` operation consists of a number of HDFS internal protocol messages (`startFile`, `getAdditionalBlock`, `writeBlockData`, and `getAdditionalBlock`). Each protocol message sent to the NameNode is handled within a single transaction. Transactions, however, only maintain consistency for individual protocol messages, and HDFS maintains consistency for the filesystem between transactions using leases on files. A lease is held throughout the duration of the HDFS filesystem operation, and in the event of a failure, the lease will eventually be released after a relatively long timeout.

We implement our transactions using the MySQL Cluster database, also known as NDB (Network Database). Read-committed is the only isolation level supported by NDB, and is a general, cheap, and widely supported transaction isolation level. Read-committed isolation means a transaction may see new state from other transactions that have completed while that transaction is executing. The read-committed isolation level allows anomalies, such as fuzzy reads, phantom reads, lost update, read skew and write skew [2], that would break the consistency of HDFS' file-system operations.

Figure 1.8. Locking in conflicting order can lead to Deadlock**Figure 1.9. Lock upgrade can lead to Deadlock***Hierarchy of Resources*

Filesystems, such as HDFS, have hierarchical namespaces, with directories containing zero to many files. In unix-based filesystems, both files and directories are represented as inodes. A filesystem hierarchy can be represented as a directed acyclic graph (DAG), when symbolic links are resolved. We ensure deadlock freedom, by ensuring that all transactions lock inodes in a consistent order, see Figure 1.8, “Locking in conflicting order can lead to Deadlock”. The consistent order is defined by the filesystem's DAG: the appropriate read or write locks are taken at inodes, starting at the root and traversing directories in a depth-first order to the leftmost leaf, continuing until either the rightmost leaf inode is reached or all the inode locks have been acquired for this transaction. The locks are then held for the duration of the transaction. Locks are not upgraded within a transaction, see Figure 1.9, “Lock upgrade can lead to Deadlock”. Since all transactions acquire locks in the same order, there are no cycles, and since locks are not upgraded, there will be no lock-upgrade deadlocks. Locks also need to be acquired on data-structures used by inodes, such as blocks, replicas, leases, corrupted-blocks, etc. In HDFS, an inode for a file contains a variable number of fix-sized blocks (typically 512 MB), where each block is typically replicated on 3 DataNodes. When operations are performed on inodes or a constituent part of an inode, we first take a write lock on the inode itself, ensuring no other transactions can concurrently access that inode or its constituent parts.

Snapshot layer

To improve the performance of our solution, we take a snapshot of the database state required for a transaction at the start of filesystem operations, store and mutate that state locally at the NameNode, and then finally commit any changes to that state at the end of the transaction, see Example 1.3, “Snapshotting at NameNodes”. This approach reduces the number of round-trips to the database, helping to improve database throughput.

Example 1.3. Snapshotting at NameNodes

```
init: snapshot.clear

operation doOperation
  tx.begin
  doSnapshot()
  performTask()
  tx.commit

operation doSnapshot
  S = total_order_sort(op.X)
  foreach x in S do
    if x is a parent then level = x.parent_level_lock
    else level = x.strongest_lock_type
    tx.lockLevel(level)
    snapshot <-tx.find(x.query)

operation performTask
  //Operation Body, accessing only state stored in the snapshot
```

For some filesystem operations, the presented algorithm is not complete enough to ensure a total order on acquiring resources. For example, there may be an inode *y* for which *x* is parent directory, and a lock on *x* should be acquired to access *y*. Since *y* is not always supplied in the operation parameter list (as is the case for many internal HDFS protocol messages) we need to first discover *y* before we can lock it. Only then, can we start the main transaction, where we lock *y* before locking *x* (based on our agreed global order for acquiring locks). We solve this problem by breaking the filesystem operation up into two transactions. In the first transaction we do not mutate state, we only resolve resources that need to be locked in the second transaction. In the second transaction, we first validate that the resources/state we resolved in the first transaction is still valid, and if so, we then perform the standard transaction, acquiring locks and taking a snapshot of the data. If the state in the second transaction is no longer valid, we can retry the whole operation, beginning at the first transaction, see Example 1.4, “Snapshotting at NameNodes using Two Transactions”.

Example 1.4. Snapshotting at NameNodes using Two Transactions

```
init: snapshot.clear, restart = true, try = 0

operation doOperation
  while restart and try < 3
    restart = false
    try = try + 1
    if op.should_resolve_parent then
      tx1.begin
      resolve_parent(op.param)
      tx1.commit

      tx2.begin
      doSnapshot()
      if data from tx1 is not valid then
        tx.abort()
        restart = true
      else
        performTask()
        tx2.commit
    end while

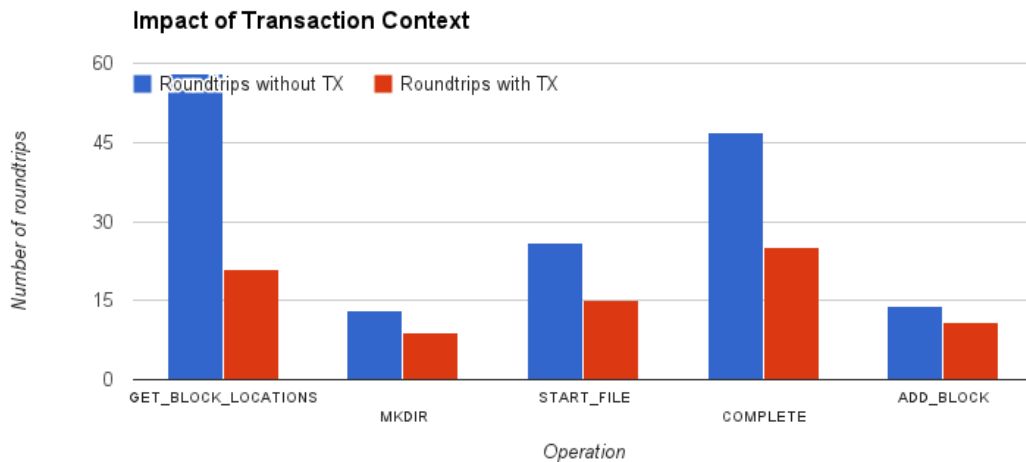
operation resolve_parent(y)
  tx.lockLevel("read_committed")
  tx.find(y.parent_query)

operation doSnapshot
  S = total_order_sort(op.X)
  foreach x in S do
    if x is a parent then level = x.parent_level_lock
    else level = x.strongest_lock_type
  tx.lockLevel(level)
  cache <- tx.find(x.query)

operation performTask
  //Operation Body, accessing only state stored in the snapshot
```

Performance of Hop HDFS

We have recently re-written our code base to migrate to HDFS 2.2. Our code is now functionally complete, but it is too early to give read/write performance figures for our new implementation. Unpublished performance figures from an earlier prototype show that Hop-HDFS can scale to handle a similar number of read and write requests per unit time as Apache HDFS, but needing additional hardware to do so. We have introduced a number of features to enable this high level of performance, including a snapshot layer at NameNodes and row-level locking at the database level, rather than a system level lock for update operations as is done in Apache HDFS. Our snapshotting layer involves a transaction acquiring all resources it requires at the start of a primitive filesystem operation, and performing local read/write operations on the snapshot copy, and then finally committing or rolling back on transaction commit.

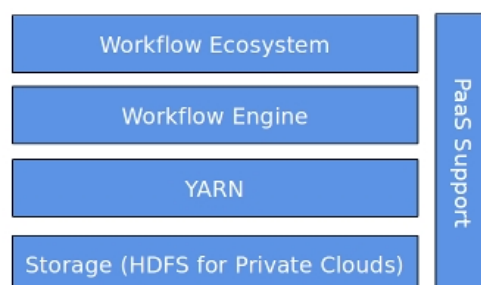
Figure 1.10. Reduction in the number of DB roundtrips by snapshotting

For improved performance, we have also implemented schema-aware partitioning of metadata across multiple hosts, ensuring all data related to a single inode hashes to the same host, and can be retrieved in a single network hop. This involved partitioning tables by inode-id, and when a transaction is started that involves a single inode, we give a hint to NDB, using the ClusterJ API, that the transaction should start on data node containing the data for that inode.

One positive aspect of HopHDFS is that the amount of metadata that can be supported is vastly increased over Apache HDFS, and it is much more easy to modify. In Apache HDFS, the NameNode is limited to the maximum size of the JVM heap for a single node, which in practice is around 100GB. In NDB, there are existing clusters of 2TB, and larger clusters are feasible. However, we have measured an expansion in the amount of memory that we require of around 43 percent. Our plan is to use this extra metadata for features such as block-level indexing, which will be very useful for BAM files used to store whole sequenced genomes, and also access control information for files. The lack of access control in HDFS is one enterprise-level feature that is preventing the use of HDFS in storing sensitive data, such as genomic data.

Hop: Platform-as-a-Service support for HDFS

HDFS exists as just one part of the larger Hadoop ecosystem for the storage and parallel processing of big data. As you can see in Figure 1.11, “Hop architecture”, our goal is to build an architecture where HDFS is the main layer for storage, YARN is used to manage the allocation of computational resources, and we support a workflow manager and workflow language for Bioinformatics (Cuneiform) as data-intensive computing support for BiobankCloud. Platform-as-a-Service (Paas) support is required at all layers in our system. Security and data sharing are not covered in this diagram, as they will be integrated in future phases of the project.

Figure 1.11. Hop architecture

1. *YARN* Yarn (Yet Another Resource Negotiator) manages the allocation of computation and memory resources to tasks in Hadoop clusters [1]. It supports many processing models, not just Map-Reduce, by separating the old JobTracker into a Resource Manager, a Scheduler and Application Master. The Application Master has the flexibility to accommodate heterogeneous processes by implementing a wrapper for each kind of application that runs over YARN, so that the application itself manages processing resources allocated to it. This enables the user to process data intensive task like MapReduce jobs or, in the BiobankCloud, to run a bioinformatics workflow engine that makes use of YARN to handle and negotiate the scheduling of its jobs.
2. *Workflow Engine* On top of YARN, the BiobankCloud workflow engine parses bioinformatic workflows written in Cuneform into an execution model of arbitrary tasks. For each task, it asks YARN for at least one container, then for each container allocated task based on the scheduling policy it stages in data into HDFS, launches the task and stages out the result, if needed. The workflow engine and language are being developed by Humboldt University as part of the BiobankCloud project.

Hop PaaS

Our PaaS for HDFS, as well as YARN and the Workflow Engine, supports the automated deployment of a Hop cluster on Amazon Cloud, Open Stack or a bare metal cluster. We call it Hadoop Open Platform-as-a-Service, or *Hop* for short, as it is *open* - designed to be deployable on any cloud platform or a bare-metal environment, not just those we currently support. Hop consists of a set of frameworks and libraries that we use to support the automated deployment of a Hadoop cluster from a website or the command-line. The main technologies we have built our prototype PaaS on are Chef and JClouds, but we are unifying all these technologies in an API based on a YAML that can be used to define a cluster that is to be deployed, see Figure 1.12, “Deploying a Hop cluster from our Portal Website”. The other technologies we currently use are BitTorrent for improving the download speeds of installation binaries and virtual machine instances (AMIs in AWS) to pre-load software onto virtual machines. The Hop PaaS API (or YAML API) can currently be accessed by a Dashboard application that we have built to manage and administer Hop clusters, see Figure 1.13, “Hop PaaS API”.

Figure 1.12. Deploying a Hop cluster from our Portal Website

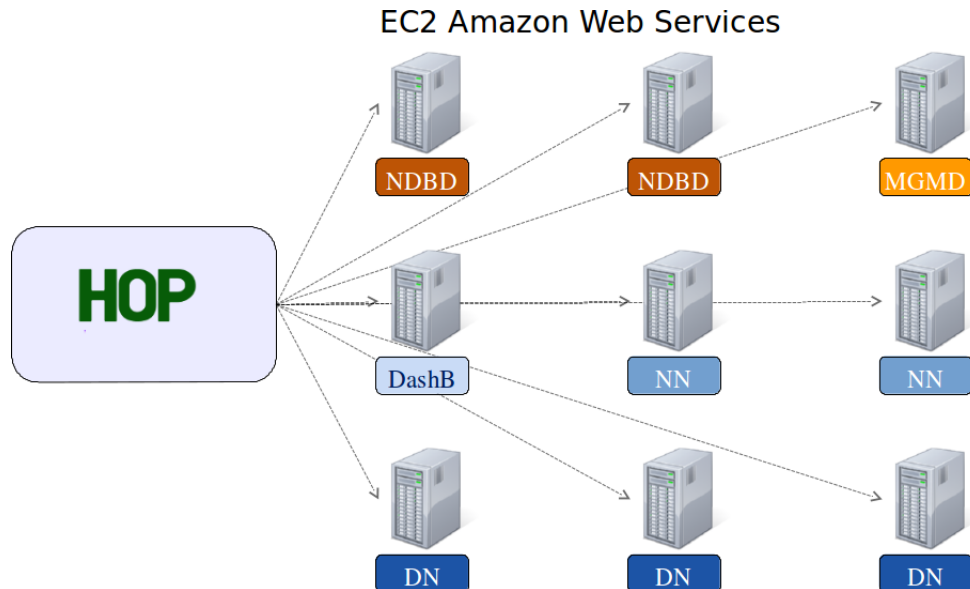
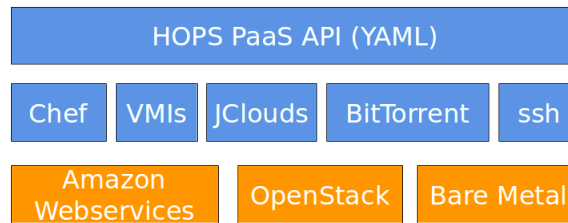


Figure 1.13. Hop PaaS API

1. **YAML** (YAML Ain't Markup Language) is a markup language which takes concepts from programming languages such as C, Perl and Python, and ideas from XML. We use YAML to define the cloud provider, the set of hosts and the services that will be installed on those hosts, together making up a single cluster. This way, we can define a whole cluster in a single file, enabling easier management of clusters and even the sharing of cluster definitions. YAML syntax allows easy mappings of common data types found in high level languages like list, associative arrays and scalar. It makes it suitable for tasks where humans are likely to view or edit data structures, such as configuration files or in our case, cluster definition files. Additionally, we make use of the open source parser SnakeYAML to parse the contents of our cluster definition files. The SnakeYAML parser transforms the given cluster definition into consecutive stages such as defining security groups, virtual machine allocation, bittorrent, installation, validation and retry. An example of a simple cluster definition is given in Example 1.5, "Example Cluster Definition in Hop". The YAML file defines a cluster consisting of 6 nodes, with MySQL Cluster running exclusively on 2 nodes, Hadoop running exclusively on 3 nodes, and another node running both MySQL Cluster and Hadoop services. The example uses AWS as the cloud provider and default values for the AWS image, instanceType and region (shown later in Chapter 5). There are many other parameters that can be overridden. Our services map directly onto chef recipes for installing the services. We are developing a model for explicitly handling recipe dependencies in chef, so that dependent services such as Java don't need to be explicitly defined as services in this cluster definition file.

Example 1.5. Example Cluster Definition in Hop

```
name: simpleCluster

provider:
  name: aws-ec2

nodes:
- services: [ndb::dn]
  number: 2

- services: [ndb::mgm, ndb::mysqld, hop::namenode]
  number: 1

- services: [hop::namenode, hop::resourcemanager]
  number: 1

- services: [hop::datanode, hop::nodemanager]
  number: 2
```

2. **Apache JClouds** Apache JClouds is an open source multi-cloud api which allows us to write reusable code for creating, destroying, and bootstrapping virtual machines (VMs) on different cloud providers. The same code can be configured to work with Amazon, OpenStack, Azure, and Rackspace VMs, and 26 other cloud providers. Through JClouds API, we can deploy and port Hop to different cloud providers. Hop parses cluster definition files, producing code that executes JCloud API calls to create, destroy and bootstrap VMs.

3. *Chef* Chef is a systems infrastructure and configuration framework that automates the deployment of applications to any physical, virtual or cloud location. When we create virtual machines (VMs) using JClouds, we receive information about the VM's IP address from the cloud provider that we use to open a ssh connection to the machine and install the Chef client. That is, JClouds installs chef during the bootstrap phase, and once the chef client is installed on a host, we can execute *chef recipes* to install software on the target host. The chef-client parses and executes a series of abstract definitions (defined as recipes inside cookbooks) written in Ruby to install software on a host. With each definition, we describe how a specific service should be installed and configured. The chef-client applies these definitions to deploy and configure applications as specified. In most of the cases, it is simple enough to let chef-client know which cookbooks and recipes it needs to apply, although recipes can be customized by users through parameterization. Although a Chef Server can be deployed to manage the chef clients, we instead use simple python agents installed at hosts and our Dashboard application to manage clients. Our chef clients install recipes using chef-solo from their local filesystem, instead of downloading recipes from the Chef Server.
4. *BitTorrent and VMIs* Chef recipe contain instructions for how to download and install software. However, some binaries are quite large, and if a large number of hosts (10s or 100s) attempt to download the same software package from the same Http Server at the same time, it can lead to congestion and slow deployment times. To handle this problem, we provide two solutions: firstly, when creating a VM image (VMI), you can specify an image that already has the software pre-installed. In AWS, this leads to fast startup times, but users lose the ability to parameterize chef recipes. In OpenStack, however, nodes all download the VMI from a Glance Http Server, so we are back to the problem of slow download times due to overloading a single server. The other option we provide is to have nodes run a bittorrent server on our Dashboard to distribute binaries among clients. The Dashboard host becomes the seeder, and all clients contribute their upload bandwidth in sharing the binaries with one another. When the binaries are downloaded, the chef recipe can be executed, configuring the software and skipping over the step of downloading the binaries. If there is a failure in the Bittorrent step, the chef recipe can download the binaries using the traditional Http server method.

References

- [1] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. "Apache Hadoop YARN: Yet Another Resource Negotiator". *2013 ACM Symposium on Cloud Computing (SoCC 2013)*. 2013. <http://hortonworks.com/blog/apache-hadoop-yarn-wins-best-paper-award-at-socc-2013/>.
- [2] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. "A critique of ANSI SQL isolation levels". *ACM SIGMOD Record*. ACM. 24. 1–10. 1995.
- [3] D Borthakur. "The hadoop distributed file system: Architecture and design". *Hadoop Project Website*. 1–14. 2007.
- [4] Dhruba Borthakur, Samuel Rash, Rodrigo Schmidt, Amitanand Aiyer, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, and Aravind Menon. "Apache hadoop goes realtime at Facebook". 1071. 2011. <http://dl.acm.org/citation.cfm?id=1989323.1989438>.
- [5] Dennis Fetterly, Maya Haridasan, Michael Isard, and Swaminathan Sundararaman. "TidyFS: a simple and small distributed file system". 34. 2011. <http://dl.acm.org/citation.cfm?id=2002181.2002215>.
- [6] Seth Gilbert and Nancy Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services". 33. 51. 2002.
- [7] Howard Gobioff, Sanjay Ghemawat, and Shun-Tak Leung. "The Google file system". 37. 29. 2003.
- [8] Haryadi S Gunawi, Abhishek Rajimwale, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. "SQCK: A Declarative File System Checker.". 131–146. 2008.

- [9] J. Gray, R. Lorie, G. Putzolu and I. Traiger. “Granularity of Locks and Degrees of Consistency in a Shared Database. In *Modeling in Data Base Management Systems*”. 365–394. 1976.
- [11] Mysql Reference Manual. “MySQL 5.0 Reference Manual”. *Syntax*. 3079. 2010.
- [12] Lev Novik, Irena Hudis, Douglas B Terry, Sanjay Anand, Vivek Jhaveri, Ashish Shah, and Yunxin Wu. “Peer-to-peer replication in WinFS”. *Technical Report MSR-TR-2006-78, Microsoft Research*. 2006.
- [13] Konstantin V Shvachko. “HDFS Scalability: The limits to growth”. *login*. 35. 6–16. 2010.
- [14] Feng Wang, Jie Qiu, Jie Yang, Bo Dong, Xinhui Li, and Ying Li. “Hadoop high availability through metadata replication”. 37–44. 2009.

Chapter 2. Quickstart with Vagrant

This section describes the steps required to deploy a Hop cluster on a single machine using git, vagrant¹, and chef².

Pre-requisites

You should have the following programs installed: git and vagrant. You will also need to download the vagrant virtual machine image for Ubuntu 12.04 "precise".

```
apt-get install git-core vagrant
vagrant box add "precise64" http://files.vagrantup.com/precise64.box
```

Launching Vagrant

You are ready to clone the chef recipes, and launch a vagrant instance.

```
git clone https://github.com/hopstart/hop-chef.git
cd hop-chef
vagrant up
```

Now grab a coffee, assuming you have a good network connection, it will take around 15 minutes to provision a vagrant instance. When vagrant successfully completes provisioning using chef, use the following URL and default user credentials to access the Hop Dashboard:

```
https://localhost:9191/hops-dashboard/
user: admin
password: admin
```

You can log into the VM and then get root access using:

```
vagrant ssh
sudo su
```

If needed, you can configure the glassfish webserver here:

```
https://localhost:5858
user: admin
password: admin
```

You can now jump to Chapter 4, *Hop Dashboard*

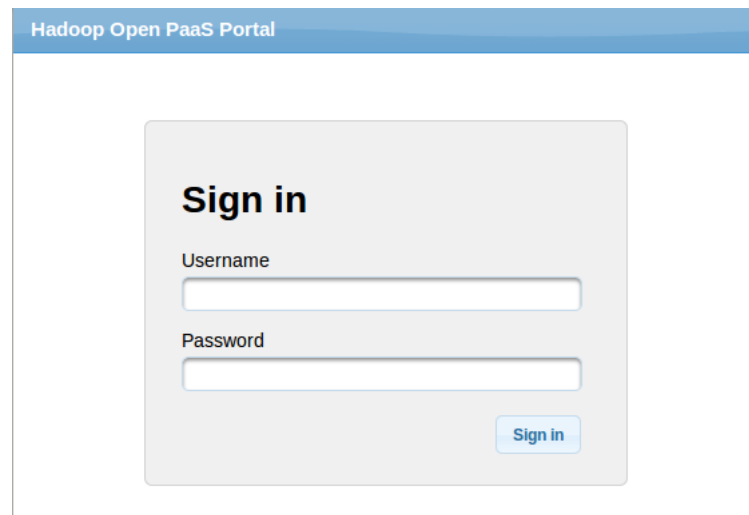
¹ Vagrant is a tool for building complete development environments. With an easy-to-use workflow and focus on automation, Vagrant lowers development environment setup time, increases development/production parity, and makes the "works on my machine" excuse a relic of the past. [http://www.vagrantup.com]

² Chef is a systems and cloud infrastructure automation framework that makes it easy to deploy servers and applications to any physical, virtual, or cloud location, no matter the size of the infrastructure. [http://docs.opscode.com/]

Chapter 3. Hop Web Portal

This section describes how to setup a Hop cluster on either a cloud provider or a baremetal cluster from our website, Hop web portal. Hop web portal is a web application where users can register, enter the credentials for their cloud provider (or ssh keys for their cluster) can from there, they can launch the Hop Dashboard on their target infrastructure. The Hop Dashboard is a web application that can be used to provision and configure the rest of the cluster. Hop Dashboard provides a centralized end-to-end management and monitoring application for Hop clusters.

Figure 3.1. Hop Portal Webiste



<https://snurran.sics.se:8181/hop-portal>

Prerequisites

The following platforms are currently supported:

- *Amazon Web Services:* You require an EC2 account credentials. The minimum recommended instance type for our Dashboard is m1.large type. Other nodes may have m1.medium or even m1.small type.
- *OpenStack:* You require credentials and configuration parameters to connect to an OpenStack end-point. The end-point must be reachable from the public Internet. The recommended instance to use in OpenStack infrastructure should be the equivalent to a m1.large instance type or greater.
- *Baremetal Cluster:* For deployment on physical hosts, you require openssh public and private keys for a user with sudo access to the hosts where you will deploy the cluster. The dashboard uses ssh to install chef on the hosts.

Currently, the only platform we support is Ubuntu 12.04.

Installing the Hop Dashboard

Intalling Hop Dashboard through the Portal is quite simple and takes a couple of minutes to deploy. Here you may find instructions on how to deploy the dashboard on AWS, OpenStack and Baremetal Machine.

Amazon EC2

Figure 3.2. AWS Credentials

The screenshot shows a 'Create Dashboard' form with three main sections:

- Dashboard Credentials:**
 - Provider: Amazon-EC2
 - Username *: Username for dashboard user
 - Password *: Password for dashboard user
- Provider Login Credentials:**
 - id *: Access Key Id from AWS
 - Secret Key: Secret Access Key for AWS
- Instance Parameters:**
 - Security Group Name *: hops
 - Hardware ID: *: m1.large
 - Image ID: *: eu-west-1/ami-ffcde8b
 - Location ID: eu-west-1
 - Enable Authorize Public Key: * ☐
 - Override Login User: ☐

A 'Launch Instance' button is located at the bottom right of the form.

To install the dashboard in Amazon EC2 follow these steps:

- Login into the HOP Portal with your user name and password.
- Select from the providers option in the maintoolbar, the Amazon EC2 option. Enter the following information in the new form
 - Dashboard credentials: admin's username and password in order to access your newly created dashboard.
 - EC2 credentials which include the Access Key id from you AWS account with its related Secret key.
 - Instance configuration parameters used to deploy a virtual machine in AWS.
 1. Security group where the machine will be deployed. If it does not exist, then a new security group will be created automatically.
 2. The hardware ID of the instance type we want to use from Amazon EC2. For example, m1.small, t1.micro. The recommended instance type is m1.large.
 3. Image ID which includes the region of that image and the *ami id* tag. We only support Ubuntu based images.
 4. Location ID of the region you want to deploy the dashboard.
 5. Selecting the option to authorize the public key will open a new option dialog box were you can insert your public key. By default we generate random key pairs for the machines through EC2 key pair service, and it is not possible to access the machines internally without this option.

6. Selecting the override login user, will override the default user for Ubuntu AMI images with the login user of your choice. This is necessary if you use custom Ubuntu images which are not one of the Ubuntu images that canonical offer in AWS by default.
- After filling up the form, press the Launch Instance button. The whole process takes 10-15 minutes. After the deployment you will receive a notification showing the URL of the newly deployed Hop Dashboard. To login the dashboard, use the credentials you specified previously in the web portal.

OpenStack

Figure 3.3. OpenStack Credentials

Create Dashboard

Dashboard Credentials

Provider: OpenStack

Username *: Username for dashboard user

Password *: Password for dashboard user

Provider Login Credentials

id *: OpenStack ProjectName:UserName

Secret Key: user password

Keystone url: keystone url of the service point

Instance Parameters

Security Group Name *: hops

Hardware ID: *: Index of the Type of Instance

Image ID: *: Id of the image from OpenStack

Location ID: Name of the Project in OpenStack

Enable Authorize Public Key: * ☐

Override Login User: * ☐

Launch Instance

IP pools in OpenStack

It is necessary that you have allocated at least 1 public IP to the project. During the deployment phase the portal will query the OpenStack project and link the public ip to the VM.

For deployment in OpenStack cloud follow these steps.

- Login into the Hop Portal with your user name and password.
- Select from the providers the OpenStack option. A new form will be generated.
- Dashboard credentials: here you specify the admin username and password for the new dashboard.
- OpenStack credentials: the user name and password to access the OpenStack project. The username should be a concatenation of the OpenStack project name and the user for that project. For example "projectName:user". Also you should indicate the url of your OpenStack Nova endpoint in order to send the requests to your OpenStack infrastructure.

- Configuration parameters that are used to deploy a virtual machine in OpenStack:
 1. Security group where the machine will be deployed. If it does not exist, we will automatically create a security group and open the ports needed for the application.
 2. The hardware ID of the instance type we want to commission in OpenStack cloud. This is a number which corresponds to the type of instance you want to deploy and is supported by your OpenStack infrastructure. We recommended using a configuration similar to a m1.large in EC2.
 3. An Image ID image located in the openstack project.
 4. Location ID identifies the dashboard in the OpenStack cluster.
 5. Selecting this option to authorize the public key based access. It will open a new dialog box where you can insert your desired public key. By default we generate random key pairs for the machine through OpenStack key pair service, and it is not possible to access the machine internally without selecting this option.
 6. Selecting the override login user: This is necessary for OpenStack if you are using a custom Ubuntu image.
- After filling up the form, press the Launch Instance button. The whole process takes 10-15 minutes. After the deployment you will receive a notification showing the URL of the newly deployed Hop Dashboard. To login the dashboard, use the credentials you specified previously in the web portal

Baremetal Hosts

Figure 3.4. Baremetal Credentials

Create Dashboard

Dashboard Credentials

Provider: Baremetal

Username *: Username for dashboard user

Password *: Password for dashboard user

SSH Credentials

host *: Machine IP or Hostname

Private Key *

Instance Parameters

Enable Authorize Public Key: ☐

Override Login User: ☐

Launch Instance

To deploy the dashboard on a BareMetal hosts cluster follow these simple steps:

- Login into the Hop Portal with your user name and password.
- In the new page, select BareMetal from the providers list. A form will be generated where you need to fill in the following:
 - Dashboard credentials: here you specify the admin username and password for the new dashboard.
 - SSH credentials: includes the host address of the machine we want to connect to and the private key.
 - Additional parameters:
 1. If you select the option to authorize a public key, it will open a new option where you can insert your desired public key to allow extra access to the machine.
 2. If you select the override login user, it will rename the sudo user to used to deploy the dashboard on the machine.
- After filling in the form, press the Launch Instance button. The whole process takes 10-15 minutes. After the deployment you will receive a notification showing the URL of the newly deployed Hop Dashboard. To login the dashboard, use the credentials you specified previously in the web portal.

Chapter 4. Hop Dashboard

The Hop Dashboard is a web application designed with Hop/Hadoop administrators in mind. The Dashboard provides a centralized end-to-end management and monitoring application for Hop clusters. It simplifies the Hadoop administrator's job by gathering all statistics for the Hop cluster in one place and presenting them in both chart and other formats. You are also able to restart many services on nodes from the Dashboard, as well as access terminals to services such as HDFS and MySQL.

After logging in to the Dashboard, press the user icon, and a menu with the following options will pop up:

- Change password
- Edit Graphs
- Backup/Restore
- Setup Credentials

Similar to Chapter 4, *Hop Dashboard* for the Hop Portal, in order to launch clusters from the dashboard, you must set your credentials for your provider: Main Menu Bar → User Icon → Setup Credentials

Graphs

The Hop Dashboard displays graphs with monitoring data for both hosts and services in your clusters. You can customize the monitoring data that is displayed by select which graphs to enable for the different services of Hop. You can also define new graphs and import the list of graphs from a file.

- *New* Selecting this option will open a new dialog where you can define the specifications for a new graph for statistics monitored by the dashboard.

Figure 4.1. Graph Editor

Model	Type	Type Instance	Data Set	Label	Color	Format	RRD File
No records found.							

- *Import* It is also possible to import a predefined set of graphs from a JSON file. This is the quickest way to load a large number of graphs.

The graph specifications are stored in a MySQL database and loaded in the dashboard. The graph table shows all the graphs for a specific type of service or a component that the dashboard is monitoring. Pressing the zoom button will generate a dialog box where you can see more details about the graph.

Figure 4.2. Graph Selection Detail

Graph							
Id	Target	Group	Plugin	Plugin Instance	Title	Vertical Label	Var
memory	HOST	Host Info	memory		Memory	Byte	

Charts							
Model	Type	Type Instance	Data Set	Label	Color	Format	RRD File
AREA	memory	used	value	Used	red	%5.2lf	memory/memory-used.rrd
AREA_STACK	memory	buffered	value	Buffered	blue	%5.2lf	memory/memory-buffered.rrd
AREA_STACK	memory	cached	value	Cached	yellow	%5.2lf	memory/memory-cached.rrd
AREA_STACK	memory	free	value	Free	green	%5.2lf	memory/memory-free.rrd

Backup/Restore

The Dashboard's state is stored entirely in a MySQL database instance. This option makes a backup of the whole dashboard state as an SQL file that can be downloaded by the user for off-site backup. In the case of failure or maintenance, the dashboard can restore a previous state from a backup SQL file.

Setup Credentials

The administrator need to specify user credentials that are used when the dashboard deploys a cluster on a cloud platform or cluster. They are typically AWS or OpenStack credentials or a public key for cluster deployments. If user credentials have not been setup, cluster deployment will fail.

Cluster Management

Cluster management is an interesting feature in the Hop Dashboard. The dashboard keeps track of different cluster applications deployed in the cloud. It allows the administrators to create, edit, delete, load and export clusters.

Figure 4.3. Manage Cluster

Cluster Management:

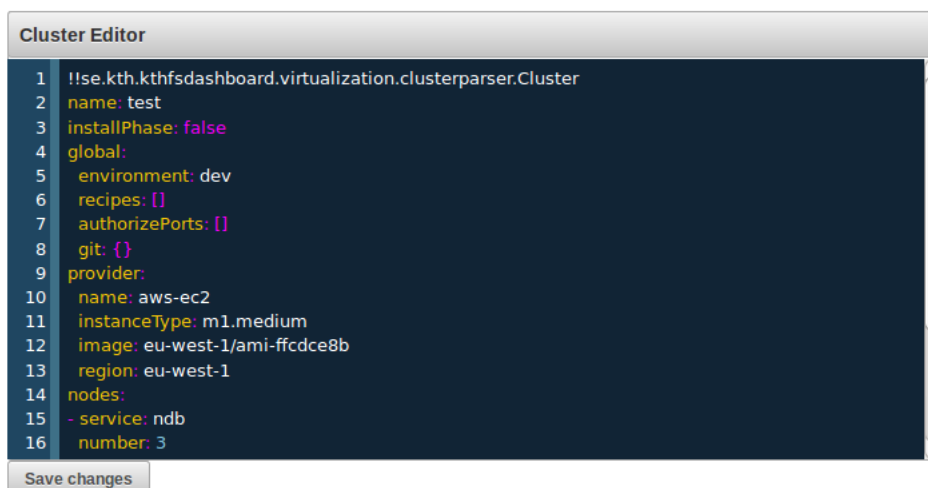
You can load from a cluster definition file No file selected.

You can also select a cluster from the database

Available cluster configurations		
Cluster Name	Cluster Type	Content
test2	virtualized	prod,aws-ec2,eu-west-1
test	virtualized	dev,aws-ec2,eu-west-1

- *Create cluster:* Selecting this option will take you to the cluster generation wizard where a users can of generate their own clusters for Hop. In the next chapter, we will explain in detail how users can create their custom clusters using our Cluster Definition Language, see Chapter 5, *Defining a Cluster*.
- *Delete clusters:* This option will delete a selected cluster. It is possible to delete multiple clusters at once by shift clicking multiple clusters before selecting this option.
- *Edit selected cluster:* This option will allow a user to edit an existing stored cluster. Administrators can use this wizard to modify the set of hosts or services in an existing cluster. They can edit the cluster either by re-running the cluster wizard or by choosing to use the embedded YAML editor available in the dashboard.

If we select the online editor option, it will open the selected cluster into a YAML editor window.

Figure 4.4. YAML Editor

- *Load selected cluster:* This option allows the user to load an existing cluster definition from a YAML file. This is a useful feature for reusing cluster definitions, users can share cluster definitions and load them using this option.
- *Export selected cluster:* It saves the information about the cluster in a YAML file, so that the cluster definition can be shared.

Cluster Deployment Progress

The Hop Dashboard monitors all the nodes deployed in its clusters. It displays the history of all the nodes going through different phases of the deployment cycle. A progress bar appears over each of the entries in the table showing the deployment progress of the nodes. A cluster node goes through the following phases during deployment:

- *Waiting:* A task has been generated in the node scheduler but no node creation query has been sent to the cloud provider.
- *Creation:* The scheduler has submitted the query to the cloud provider and it is waiting for the cloud provider to finish deploying the virtual instance. When an instance is successfully created an initialization script is executed for preliminary configuration of the node.
- *Install:* After the node is successfully commissioned, chef installs recipes for Hop Services to fetch the necessary binaries from the different repositories. This phase is optional in case of using pre built virtual instances which contain Hop binaries.
- *Configure:* A node in this phase means, that; the node is receiving the configuration script which will execute chef with the selected recipes for the services defined for that node.
- *Complete:* The node has successfully finished executing the configuration script with chef and it is now part of the working cluster.
- *Retrying:* The deployment system has detected a problem during a node deployment phase and it is retrying to recover from the failure. It retries the failed phase script for five times.
- *Error:* When all the retries are exhausted the node is marked erroneous. Hadoop administrator can take further actions to recover the node, for example, SSH into the failed node and the fix the problem manually, retry the deployment process or delete the failed nodes.

Monitoring

The Hop Dashboard monitors hosts and services and also displays alerts in real time. A brief overview of the monitoring functionality is given below.

Hosts

The host monitoring tool shows the state of all the nodes in the clusters. An administrator can track information of his/her interest such as the allocated IPs for the nodes, host names, host IDs, health of the nodes, and when the last heartbeat was received from the nodes. It also monitors available resources on the nodes such as the number of cores that machine has, average load on the instance, disk usage and physical memory in use.

Figure 4.5. Hosts

Hadoop Open PaaS Dashboard


Clusters

Hosts

Alerts

Manage Clusters

Clusters Progress

 admin

Hosts

1 Host Under Management:

Add Host

Hosts

Host Id	Hostname	Public IP	Private IP	Health	Last Heartbeat	Cores	Load Average			Disk Usage	Physical Memory
10	mgmd49		10.0.2.15	Good	4.9s ago	2	0.92	1.15	0.85	<div><div></div>10 GB / 78.9 GB</div>	<div><div></div>2.4 GB / 3.8 GB</div>

Selecting a host displays a set of services installed on the host, Figure 4.6, “Hosts Services”, and a set of graphs monitoring the host's performance, Figure 4.7, “Host Graphs”.

Figure 4.6. Hosts Services

Hadoop Open PaaS Dashboard

Clusters

Hosts

Alerts

Manage Clusters

Clusters Progress

admin

Hosts »

10 (mgmd49)

Status

Host Details

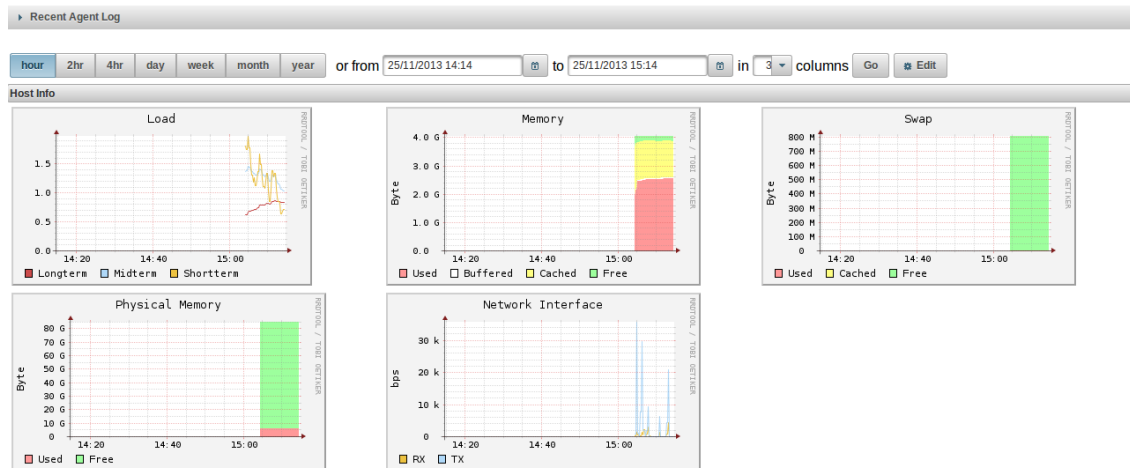
Host Id	Hostname	Public IP	Private IP	Health	Last Heartbeat	Cores	Load Average			Disk Usage	Physical Memory
10	mgmd49		10.0.2.15	Good	1.3s ago	2	0.94	1.07	0.85	<div><div></div>10 GB / 78.9 GB</div>	<div><div></div>2.4 GB / 3.8 GB</div>

Roles

Cluster	Service	Role	Role Page
vagrant	HOPS	datanode	vagrantHOPS/datanode @ 10
vagrant	HOPS	namenode	vagrantHOPS/namenode @ 10
vagrant	MySQLCluster	memcached	vagrantMySQLCluster/memcached @ 10
vagrant	MySQLCluster	mgmserver	vagrantMySQLCluster/mgmserver @ 10
vagrant	MySQLCluster	mysqld	vagrantMySQLCluster/mysqld @ 10
vagrant	MySQLCluster	ndb	vagrantMySQLCluster/ndb @ 10
vagrant	YARN	NodeManager	vagrantYARN/NodeManager @ 10
vagrant	YARN	ResourceManager	vagrantYARN/ResourceManager @ 10

Recent Agent Log

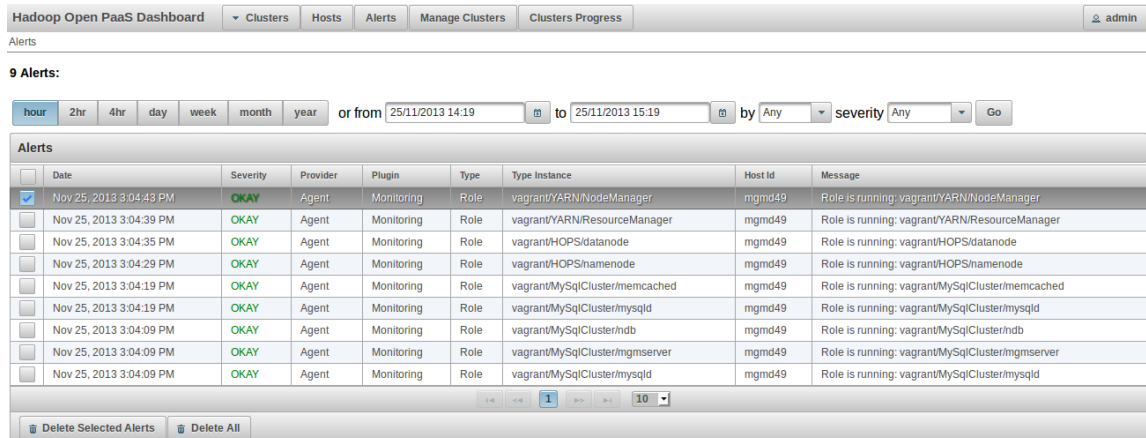
Figure 4.7. Host Graphs



Alerts

You can define alerts as thresholds in collectd. Examples of thresholds would be cpu at 100%, hard drives at 95% capacity, etc. From the Dashboard, you can filter alerts both by severity and within a given time-window.

Figure 4.8. Alerts



Clusters

In the Clusters tab you view and manage the set of deployed clusters, as well as create new clusters. Here, you can see information about the nodes that compose the cluster and the health status of the services deployed on those nodes. You can drill down into a cluster to find out the list of services in the cluster, their status and a list of nodes for each service. Many of the services can be restarted from the Dashboard in the event of failure.

Figure 4.9. Clusters

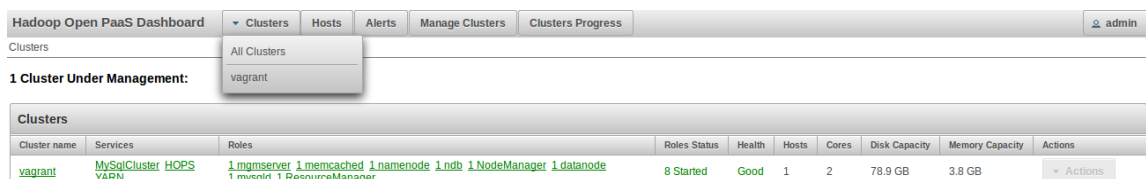
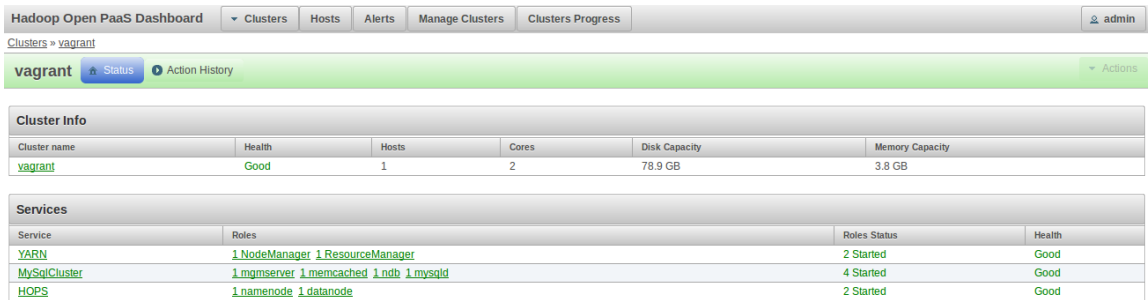
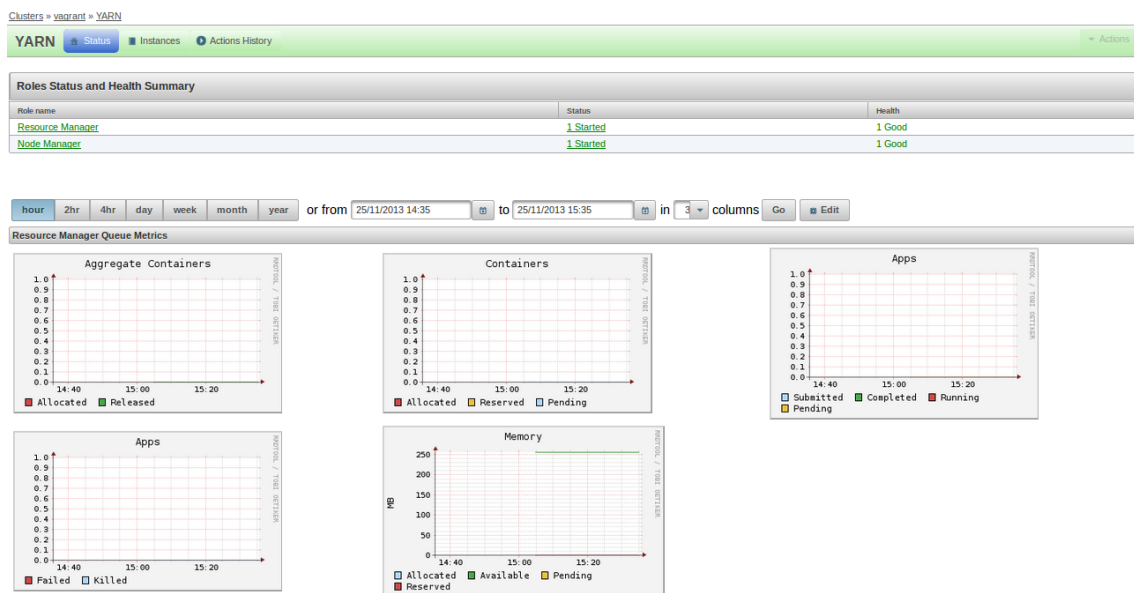


Figure 4.10. Detailed Cluster View

YARN monitoring

The Hop Dashboard displays graphs on YARN performance statistics.

Figure 4.11. YARN Metrics

You can also access performance statistics for the resourcemanager and nodemanagers in YARN.

Figure 4.12. Resource Manager Metrics

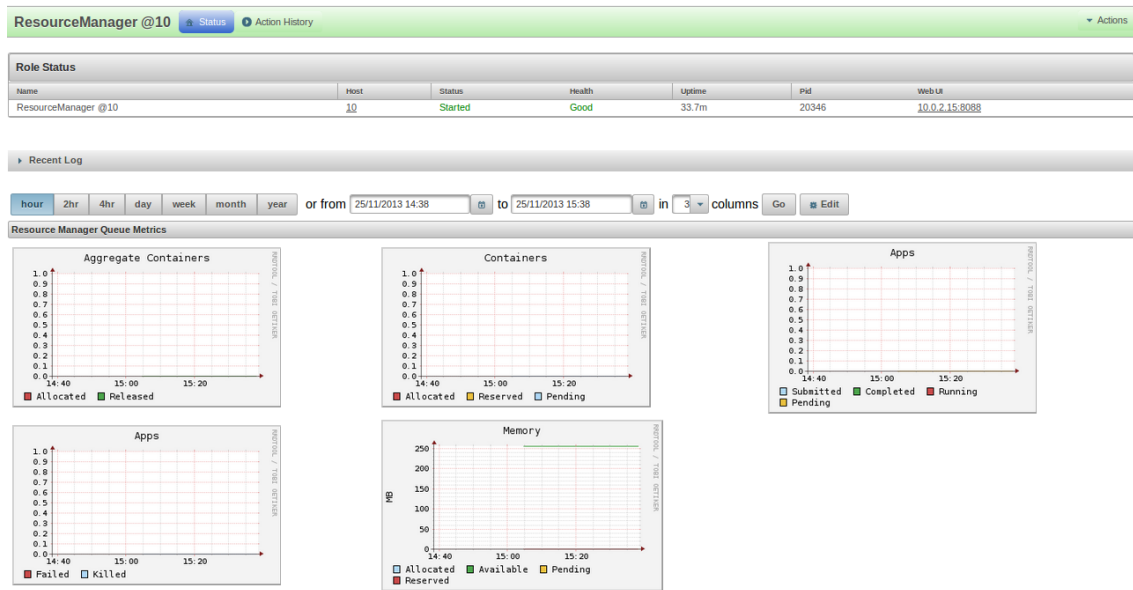
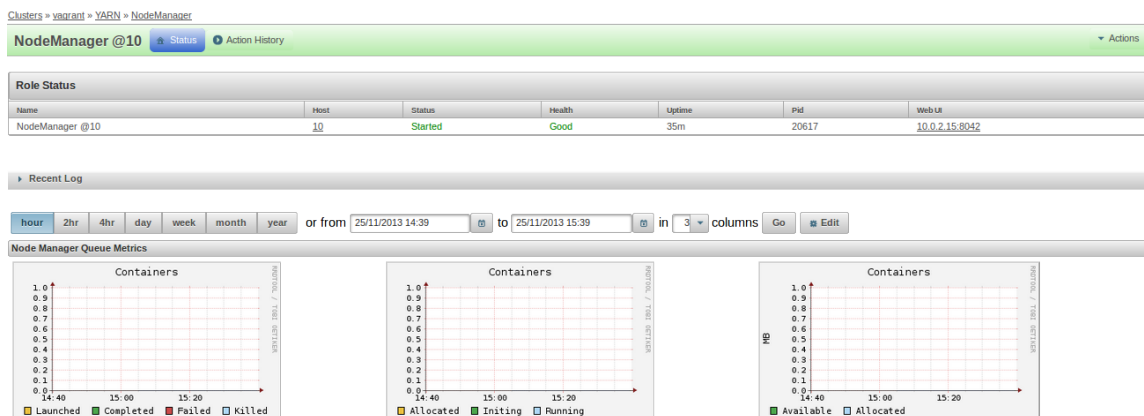


Figure 4.13. Node Manager Metrics



The Hop dashboard provides direct access to the resourcemanager and nodemanager web UIs provided by the Apache distribution. The resource manager web UI gives detailed information about the number of applications submitted, number of applications in progress, number of application completed, total available memory, memory consumed, dead nodes etc. The Node manager web UI displays information such as total memory allocated to the java virtual machine on the node, memory allocated to each container, node health status, Hadoop version, etc.

Figure 4.14. Resource Manager UI

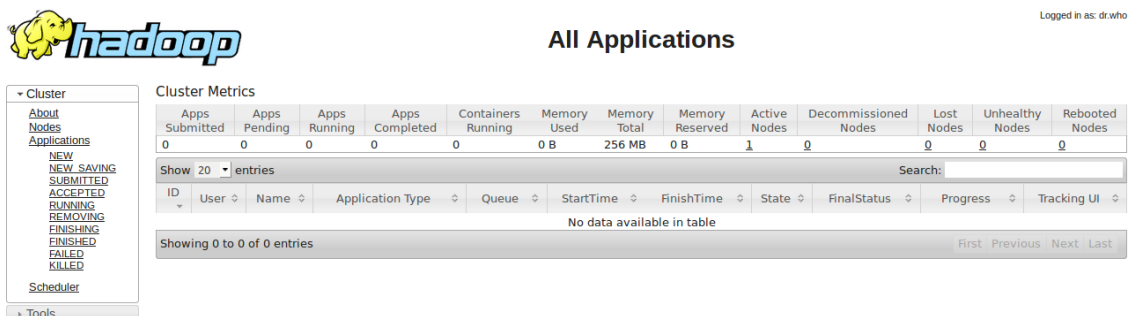


Figure 4.15. Node Manager UI



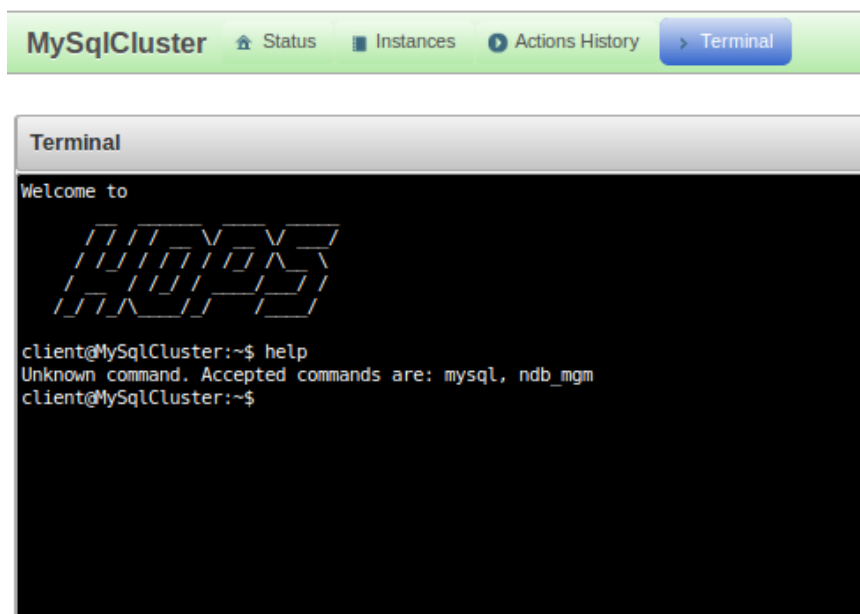
MySQL Cluster

You can monitor the MySQL Cluster using the Hop Dashboard. It keeps track of data memory, index memory, number of reads and writes per second and number of scans per second. Data nodes, management nodes, and MySQL servers can all be restarted from the Dashboard.

Figure 4.16. MySQL Cluster Monitor

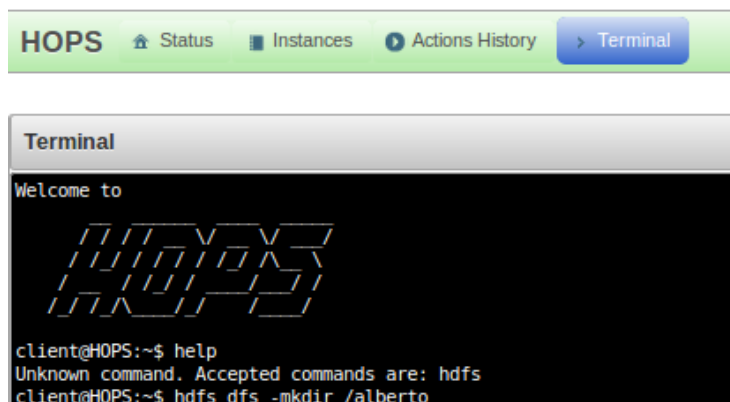


The dashboard provides a command-line terminal to directly run SQL commands on a MySQL Server connected to NDB.

Figure 4.17. MySQL Console

Hop HDFS Console

The dashboard provides a command-line terminal for executing HDFS operations using the HDFS client. It performs operations such as ls, mkdir, cat, etc.

Figure 4.18. Hdfs Console

Chapter 5. Defining a Cluster

Hops allows you to define a cluster in a single YAML file. A cluster definition consists of a provider (cloud or bare-metal), a number of nodes and a set of services deployed on those nodes. The benefit of having a cluster defined in a single file is that you can now share it with others, who launch the same cluster or modify the file and launch a changed version of your cluster, and you can edit it to reconfigure your cluster.

A cluster definition file consists of the following sections:

- [Global]
- [Provider]
- Nodes
 - Services
 - Number|Hosts

In the example cluster definition file presented in Chapter 1, we showed a minimal cluster definition, that had some default values not specified in the cluster definition file. There was no *Global* section, and it wasn't stated where the implementations of the services came from. In Example 5.1, “Complete Example Cluster Definition in Hop”, we give the complete version of the cluster definition file. We can see that by default, `ssh`, `chef`, and `collectd` are installed on all nodes, and that the chef recipes for installing the services are taken from <https://github.com/hopstart/hop-chef.git>. Users can fork the chef recipes we provide, customizing and storing them in their own git repository. Soon, we will provide support for users specifying their own recipes as

The file below also shows the full set of defaults for our AWS instances. Currently, we only support compute instances. Each services section can be further customized with a `chefAttributes` section, containing parameters for the chef recipes.

Example 5.1. Complete Example Cluster Definition in Hop

```
!!se.kth.dashboard.virtualization.clusterparser.Cluster
name: simpleCompleteCluster

global:
  services: [ssh, chef, collectd]
  authorizePorts: [22, 4343, 3321]
  git-user: jdowling
  git-repository: https://github.com/hopstart/hop-chef.git
  git-key: /home/jdowling/.ssh/id_rsa

provider:
  name: aws-ec2
  instanceType: m1.large
  loginUser: ubuntu
  image: eu-west-1/ami-35667941
  region: eu-west-1

nodes:
  - services:
    - ndb::dn
    number: 2

  - services:
    - ndb::mgm
    number: 1

  - services:
    - ndb::mysqld
    - hop::namenode
    number: 1

  - services:
    - hop::namenode
    - hop::resourceManager
    number: 1

  - services:
    - hop::datanode
    - hop::nodemanager
    number: 2
```

It is possible to take the above cluster definition and deploy the same set of VMs and services on an OpenStack cluster. To do this, we need to modify the Provider section. *Provider:* You need to specify the infrastructure provider, the image you want to use, the type of instance to request, and login credentials in case you are using custom images. In the example below, we show how to change the provider section to prepare the cluster definition file to work for OpenStack.

Example 5.2. OpenStack provider

```
provider:
name: openstack-nova
instanceType: 7
loginUser: ubuntu
image: 0190f9c4-d64e-4412-ab88-4f9fd1d7c2e3
region: RegionSICS
```

We can also deploy the same cluster on bare-metal hosts in a cluster. In this case, you need to provide the IP addresses of the machines where the services will be installed. Deploying on bare-metal requires specifying a new `clusterparser` (`se.kth.dashboard.virtualization.clusterparser.Baremetal`) as well as removing the provider section, and specifying IP addresses in the services sections, instead of number of nodes.

Example 5.3. Bare-metal cluster

```
!!se.kth.dashboard.virtualization.clusterparser.Baremetal
name: baremetal
loginUser: ubuntu

nodes:
- services:
- ndb::dn
hosts:
- 10.20.0.8
- 10.20.0.11

- services:
- ndb::mgm
hosts:
- 10.20.0.6

- services:
- ndb::mysqld
- hop::namenode
hosts:
- 10.20.0.7

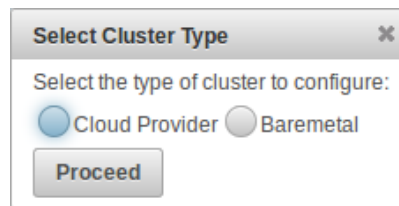
- services:
- hop::namenode
- hop::resourcemanager
hosts:
- 10.20.0.12
- 10.20.0.14

- services:
- hop::datanode
- hop::nodemanager
hosts:
- 10.20.0.16
- 10.20.0.17
```


Cluster Wizard

In addition to being able to define a cluster in YAML, you can also generate a cluster using a wizard with the following these steps:

1. Press the create cluster button following the menu items: Main Menu Bar → Manage Cluster → Create cluster
2. A dialog appears allowing you to select which type of cluster you want to use:
 - *Virtualized*: Choose this option if you want to deploy a cluster in Amazon EC2 or OpenStack.
 - *Baremetal*: Choose this option if you want to deploy a cluster in physical machines.



3. Selecting an option, will bring you to the cluster generator wizard. Here you can select the same options like if you where writing your own file from scratch. You will go through different phases.

Cluster Wizard → Common → Provider (not for Baremetal) → Groups → Confirmation

- *Common Section*: In this section, a form appears were you can select the following options:
 - a. *Name*: Name of the cluster
 - b. *Provider*: Select the type provider between Amazon EC2 or OpenStack, this option is available if we create a virtualized cluster.
 - c. *Git parameters*: Git repository section where you can specify as an option your own git repository based on our code. This way you can customize our recipes or even add your own.
 - d. *Global Recipes*: You can specify chef recipes that you want to execute in all the nodes
 - e. *Global Ports*: Additional Ports to open for your cluster, this option is only available for virtualized clusters.

Figure 5.1. Common Cluster Options:

Common

Provider

Groups

Confirmation

Global Attributes

Cluster name: * test2
Provider: * Amazon
Environment: * Production
Install Phase:

Git parameters (Optional)

Global Recipes

Click delete selection after selecting multiple instances to delete, use shift-click to select multiples

Recipe	Actions
No records found.	

+ New Recipe
Delete Selection

Global Ports

Click delete selection after selecting multiple instances to delete, use shift-click to select multiples

Port Number	Actions
3306	
4343	
3321	

+ New Port
Delete Selection

Next

Figure 5.2. Bare Metal Common Cluster Options:

Common

Groups

Confirmation

Global Attributes

Cluster name: * baremetal
Login user: * ubuntu
Total number of Hosts: * 8
Environment: * Production
Install Phase:

Git parameters (Optional)

Global Recipes

Click delete selection after selecting multiple instances to delete, use shift-click to select multiples

Recipe	Actions
No records found.	

+ New Recipe
Delete Selection

Next

- *Provider Section:* This form enables you to define the parameters for OpenStack or Amazon EC2. Some values appear by default in the case of Amazon EC2 of defining a cluster to be used with this cloud provider.

- a. *Instance Type*: The type of instance you want to use in Amazon EC2 or in OpenStack. Note that in OpenStack we use the id number of the type of instance, not the name.
- b. *Image*: The name of the image we want to use the in Amazon EC2 or in OpenStack
- c. *Login user*: Here you include the user name with sudo access to access the instances in Amazon EC2 or OpenStack. Note that this value is necessary if you use a custom AMI in Amazon EC2 or using you use OpenStack.
- d. *Region*: Here you include the region you want to deploy in Amazon EC2 or the project to use in your OpenStack infrastructure.

Figure 5.3. Cluster Provider Options:

- *Group Section*: In this section you can specify the group of nodes for you cluster with the their services and ip addresses (if you are deploying a baremetal cluster)
 - a. *Main Service*: The main service you want to deploy in this group of nodes
 - b. *Bittorrent Support*: If you want to enable bittorrent sync of binaries from the dashboard.
 - c. *Number of nodes*: Number of nodes that will contain the same set of services.
 - d. *Extra Services*: Other services you may want to run which can be also your own services.
 - e. *ChefAttributes*: In this section, you would include a chef json which will contain the attributes you may want to override from your recipes.
 - f. *Ports*: Extra ports that you may want to enable in that group, in this case this only affect virtualized clusters.
 - g. *Hosts*: List of hosts IP addresses for the nodes that will be part of this group of nodes. In this case this option is only available for Baremetal clusters.

Figure 5.4. Cluster Group:

Common Provider **Groups** Confirmation

Cluster Nodes

Click delete selection after selecting multiple instances to delete, use shift-click to select multiples

Service	Number of Nodes	Recipes	Ports	Bittorrent	Chef Attributes
ndb	2				
mgm	1				
mysqld	1				
namenode	2				
datanode	2				

+ New Group Edit selection - Delete Selection

Back Next

Figure 5.5. Bare Metal Groups:

Common **Groups** Confirmation

Cluster Nodes

Click delete selection after selecting multiple instances to delete, use shift-click to select multiples

Service	Number of Nodes	Recipes	Hosts:	Bittorrent	Chef Attributes
ndb	2	[MySQLCluster-ndb]	[10.20.0.8, 10.20.0.11]		
mgm	1	[MySQLCluster-mgm]	[10.20.0.6]		
mysql	1	[MySQLCluster-mysqld]	[10.20.0.7]		
namenode	2	[KTHFS-namenode]	[10.20.0.12, 10.20.0.14]		
datanode	2	[KTHFS-datanode]	[10.20.0.16, 10.20.0.17]		

+ New Group Edit selection - Delete Selection

Back Next

- *Confirmation Section:* In this section you will see a summary of the details of your cluster file. When you press the submit button, your cluster file will be stored in the dashboard and it will proceed to the cluster launcher.

Figure 5.6. Confirmation

Common

Provider

Groups

Confirmation

Confirmation

▼ General Details:

Name:

test2

Environment:

prod

Install Phase:

false

Authorize Ports:

[3306, 4343, 3321]

Git user:

Jim Dowling

Git repository:

https://ghetto.sics.se/jdowling/kthfs-pantry.git

Git key:

notNull

▶ Provider Details:

▶ Nodes:

Submit

← Back

Chapter 6. Configuring HDFS

We introduce a few new configuration parameters to HDFS, due to our support for multiple NameNodes and use of MySQL Cluster for metadata storage. These parameters are specified in *hdfs-site.xml*. The configuration parameters listed below are additional to the configuration parameters for vanilla HDFS [<http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/hdfs-default.xml>].

HDFS Configuration Parameters not used

We have replaced HDFS 2.x's Primary-Secondary Replication model with shared atomic transactional memory. This means that we no longer use the parameters in HDFS that are based on the (eventually consistent) replication of *edit log entries* from the Primary NameNode to the Secondary NameNode using a set of quorum-based replication servers. Here are the parameters that are not used in the HOP version of HDFS 2.x:

- *dfs.namenode.secondary.**: None of the secondary NameNode attributes are used.
- *dfs.namenode.checkpoint.**: None of the checkpoint attributes are used.
- *dfs.image.**: None of the FSImage attributes are used.
- *dfs.journalnode.**: None of the hadoop's journaling attributes are used.
- *dfs.ha.**: None of the hadoop high availability attributes are used.
- *dfs.namenode.num.extra.edits.**: None of the edit logs attributes are used.
- *dfs.namenode.name.dir.**: FSImage is not supported anymore.
- *dfs.namenode.edits.**: None of the edit log attributes are used.
- *dfs.namenode.shared.edits.**: None of the edit log attributes are used.

Additional HDFS Configuration Parameters

- *dfs.storage.type*: In HOP all the NameNodes in the system are stateless. All the file system metadata is stored in a relational database. We have chosen MySQL NDB Cluster for its high performance and availability for the storage of the metadata. However the metadata can be stored in any relational database. Default value is this parameter is 'clusterj'. By default HOPS uses ClusterJ libraries to connect to MySQL NDB Cluster. Later we will provide support of other DBMSs.
- *dfs.dbconnector.string*: Host name of management server of MySQL NDB Cluster.
- *dfs.dbconnector.database*: Name of the database that contains the metadata tables.
- *dfs.dbconnector.num-session-factories*: This is the number of connections that are created in the ClusterJ connection pool. If it is set to 1 then all the sessions share the same connection; all requests for a SessionFactory with the same connect string and database will share a single SessionFactory. A setting of 0 disables pooling; each request for a SessionFactory will receive its own unique SessionFactory. We set the default value of this parameter to 3.
- *dfs.storage.mysql.user*: A valid user name to access MySQL Server. For higher performance we use MySQL Server to perform aggregate queries on the file system metadata.
- *dfs.storage.mysql.user.password*: MySQL user password
- *dfs.storage.mysql.port*: MySQL Server port. If not specified then default value of 3306 is chosen.

- *dfs.quota.enabled*: Using this parameter quota can be en/disabled. By default quota is enabled.
- *dfs.namenodes.rpc.address*: HOP support multiple active NameNodes. A client can send a RPC request to any of the active NameNodes. This parameter specifies a list of active NameNodes in the system. The list has following format [ip:port, ip:port, ...]. It is not necessary that this list contain all the active NameNodes in the system. Single valid reference to an active NameNode is sufficient. At the time of startup the client will obtain the updated list of all the NameNodes in the system from the given NameNode. If this list is empty then the client will connect to 'fs.default.name'.
- *dfs.namenode.selector-policy*: For a RPC call client will choose an active NameNode based on the following policies.

1. ROUND_ROBIN

2. RANDOM

By default NameNode selection policy is set of ROUND_ROBIN

- *dfs.leader.check.interval*: One of the active NameNodes is chosen as a leader to perform housekeeping operations. All NameNodes periodically send a HeartBeat and check for changes in the membership of the NameNodes. By default the HeartBeat is sent after every second. Increasing the time interval would lead to slow failure detection.
- *dfs.leader.missed.hb*: This property specifies when a NameNode is declared dead. By default a NameNode is declared dead if it misses a HeatBeat. Higher values of this property would lead to slow failure detection.
- *dfs.block.pool.id*: Due to shared state among the NameNodes, HOP only support one block pool. Set this property to set a custom value for block pool. Default block poold id is HOP_BLOCK_POOL_123.
- *dfs.name.space.id*: Due to shared state among NameNodes, HOP only support one name space. Set this property to set a custom value for name space. Default name space id is 911 :)
- *dfs.clinet.max.retires.on.failure*: The client will retry the RPC call if the RPC fails due to the failure of the NameNode. This property specifies how many times the client would retry the RPC before throwing an exception. This property is directly related to number of expected simultaneous failures of NameNodes. Set this value to '1' in case of low failure rates such as one dead NameNode at any given time. It is recommended that this property must be set to value ≥ 1 .
- *dsf.client.max.random.wait.on.retry*: A RPC can fail because of many factors such as NameNode failure, network congestion etc. Changes in the membership of NameNodes can lead to contention on the remaining NameNodes. In order to avoid contention on the remaining NameNodes in the system the client would randomly wait between [0,MAX_VALUE] ms before retrying the RPC. This property specifies MAX_VALUE; by default it is set to 1000 ms.
- *dsf.client.refresh.namenode.list*: All clients periodically refresh their view of active NameNodes in the system. By default after every minute the client checks for changes in the membership of the NameNodes. Higher values can be chosen for scenarios where the membership does not change frequently.

Chapter 7. Conclusions

In this document, we presented our highly available HDFS architecture. Our version of HDFS provides a new model for stateless NameNodes, with metadata stored in highly available shared, transactional memory, implemented using MySQL Cluster. We have solved problems related to leader election, maintaining consistency of the filesystem operations after migrating metadata to a relational store, and snapshotting of database state to reduce the number of database roundtrips, thus improving throughput. We have also presented Hop (Hadoop Open Paas), our platform-as-a-service for automating the deployment of clusters, as well as a language for describing clusters. Finally, we included a userguide to get started using Hop.