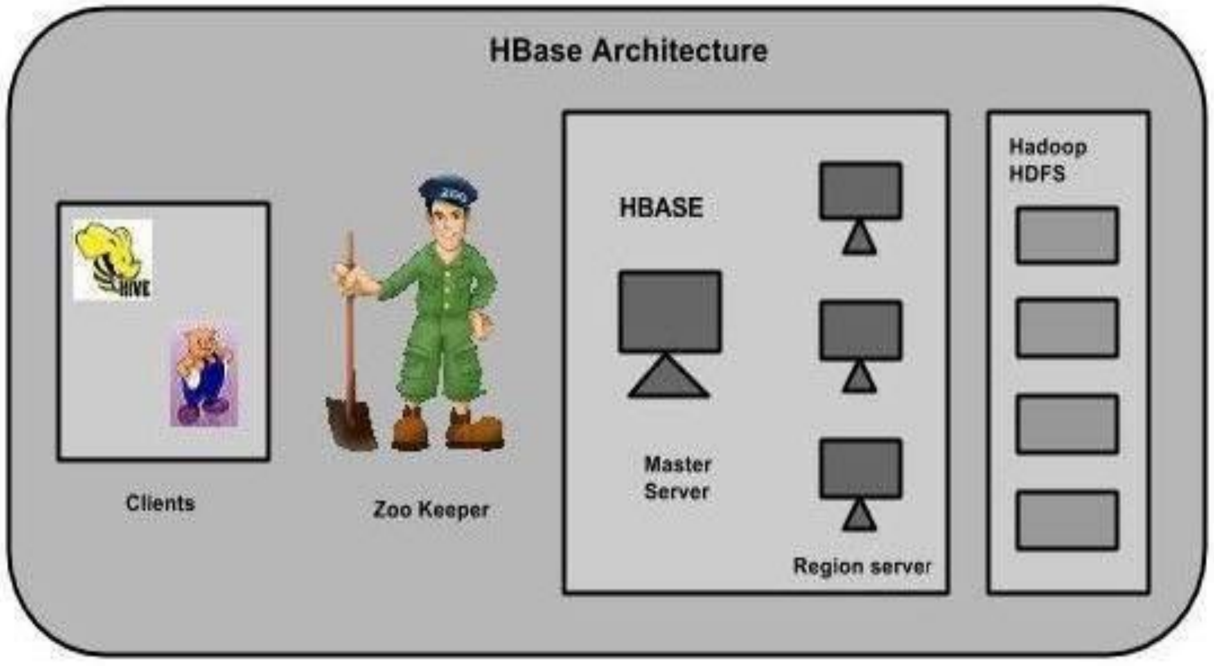
UNIT-V Hadoop Project Environment

HBASE-ARCHITECTURE

In HBase, tables are split into regions and are served by the region servers. Regions are vertically divided by column families into “Stores”. Stores are saved as files in HDFS. Shown below is the architecture of HBase.

Note: The term ‘store’ is used for regions to explain the storage structure.



HBase has three major components: the client library, a master server, and region servers. Region servers can be added or removed as per requirement.

MasterServer

The master server -

- Assigns regions to the region servers and takes the help of Apache ZooKeeper for this task.
- Handles load balancing of the regions across region servers. It unloads the busy servers and shifts the regions to less occupied servers.
- Maintains the state of the cluster by negotiating the load balancing.
- Is responsible for schema changes and other metadata operations such as creation of tables and column families.

Regions

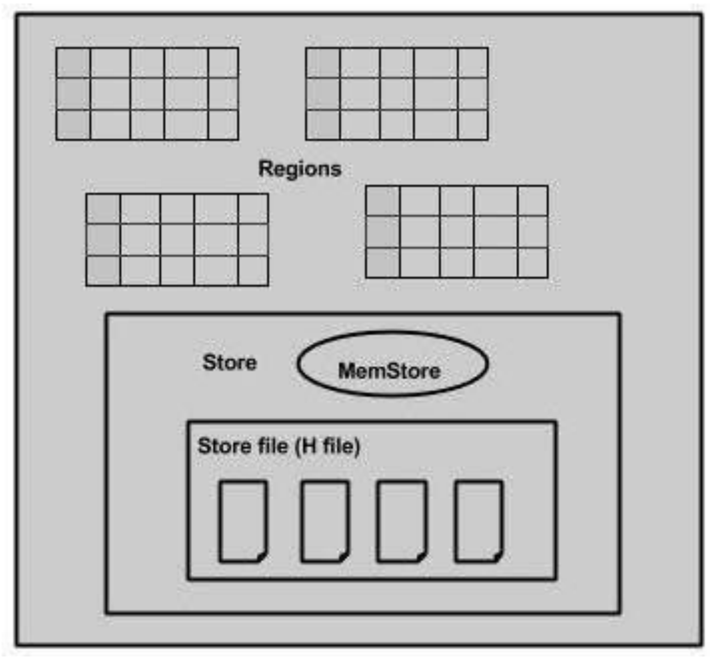
Regions are nothing but tables that are split up and spread across the region servers.

Region server

The region servers have regions that -

- Communicate with the client and handle data-related operations.
- Handle read and write requests for all the regions under it.
- Decide the size of the region by following the region size thresholds.

When we take a deeper look into the region server, it contain regions and stores as shown below:



The store contains memory store and HFiles. Memstore is just like a cache memory. Anything that is entered into the HBase is stored here initially. Later, the data is transferred and saved in Hfiles as blocks and the memstore is flushed.

Zookeeper

- Zookeeper is an open-source project that provides services like maintaining configuration information, naming, providing distributed synchronization, etc.
- Zookeeper has ephemeral nodes representing different region servers. Master servers use these nodes to discover available servers.

- In addition to availability, the nodes are also used to track server failures or network partitions.
- Clients communicate with region servers via zookeeper.
- In pseudo and standalone modes, HBase itself will take care of zookeeper.

Client API Features:-Filters

HBase filters are a powerful feature that can greatly enhance your effectiveness when working with data stored in tables.

The two prominent read functions for HBase are get() and scan(), both supporting either direct access to data or the use of a start and end key, respectively.

These include column families, column qualifiers, timestamps or ranges, as well as version number.

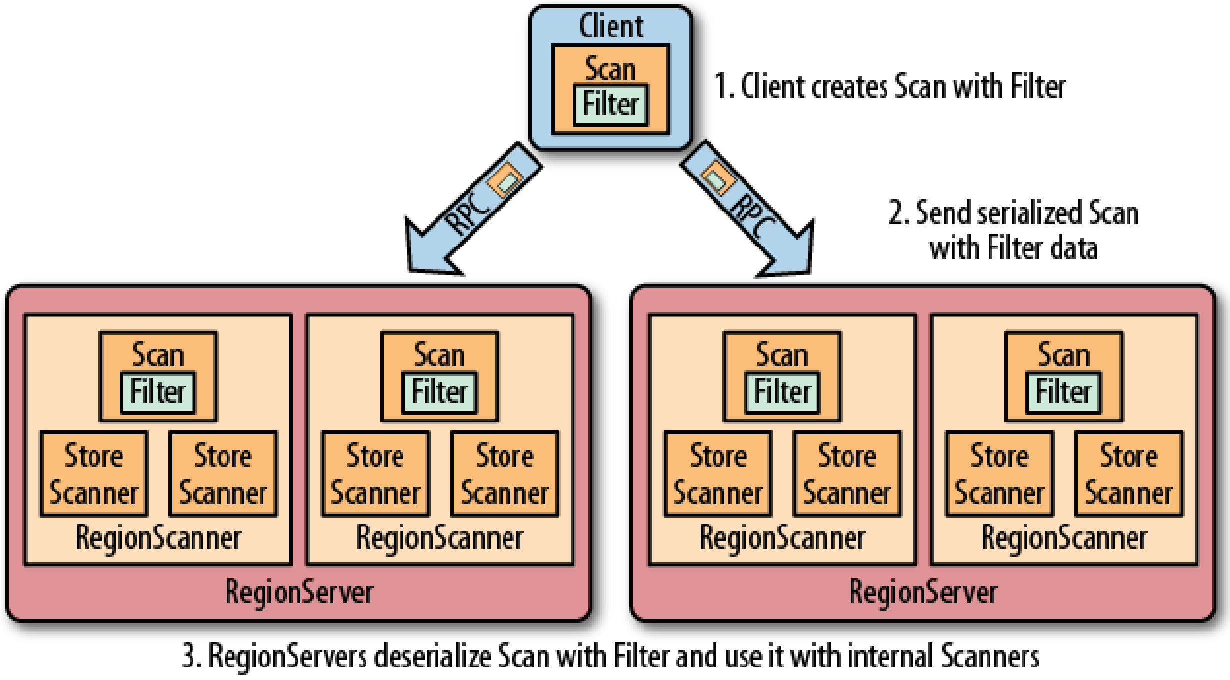

Client
Scan
Filter
1. Client creates Scan with Filter
RPC
RPC
2. Send serialized Scan with Filter data
Scan
Filter
Store Scanner
Store Scanner
RegionScanner
RegionServer
3. RegionServers deserialize Scan with Filter and use it with internal Scanners

Fig- Shows the filters created on the client side, sent through the RPC, and executed on the server side

The filter hierarchy

The lowest level in the filter hierarchy is the Filter interface, and the abstractFilterBase class that implements an empty shell, or skeleton, that is used by the actual filter classes to avoid having the same boilerplate code in each of them.

You define a new instance of the filter you want to apply and hand it to the Get or Scan instances, using:

```
setFilter(filter)
```

Comparison operators

As CompareFilter-based filters add one more feature to the base FilterBaseclass, namely the compare() operation, it has to have a user-supplied operator type that defines how the result of the comparison is interpreted.

Table The possible comparison operators for CompareFilter-based filters

Operator	**Description**
LESS	Match values less than the provided one.
LESS_OR_EQUAL	Match values less than or equal to the provided one.
EQUAL	Do an exact match on the value and the provided one.
NOT_EQUAL	Include everything that does not match the provided value.
GREATER_OR_EQUAL	Match values that are equal to or greater than the provided one.
GREATER	Only include values greater than the provided one.
NO_OP	Exclude everything.

The comparison operators define what is included, or excluded, when the filter is applied. This allows you to select the data that you want as either a range, subset, or exact and single match.

Comparators

The second type that you need to provide to CompareFilter-related classes is a *comparator*, which is needed to compare various values and keys in different ways. They are derived from WritableByteArrayComparable, which implements Writable, and Comparable. You do not have to go into the details if you just want to use an implementation provided by HBase and listed in the Table . The constructors usually take the control value, that is, the one to compare each table value against.

Table- The HBase-supplied comparators, used with CompareFilter-based filters

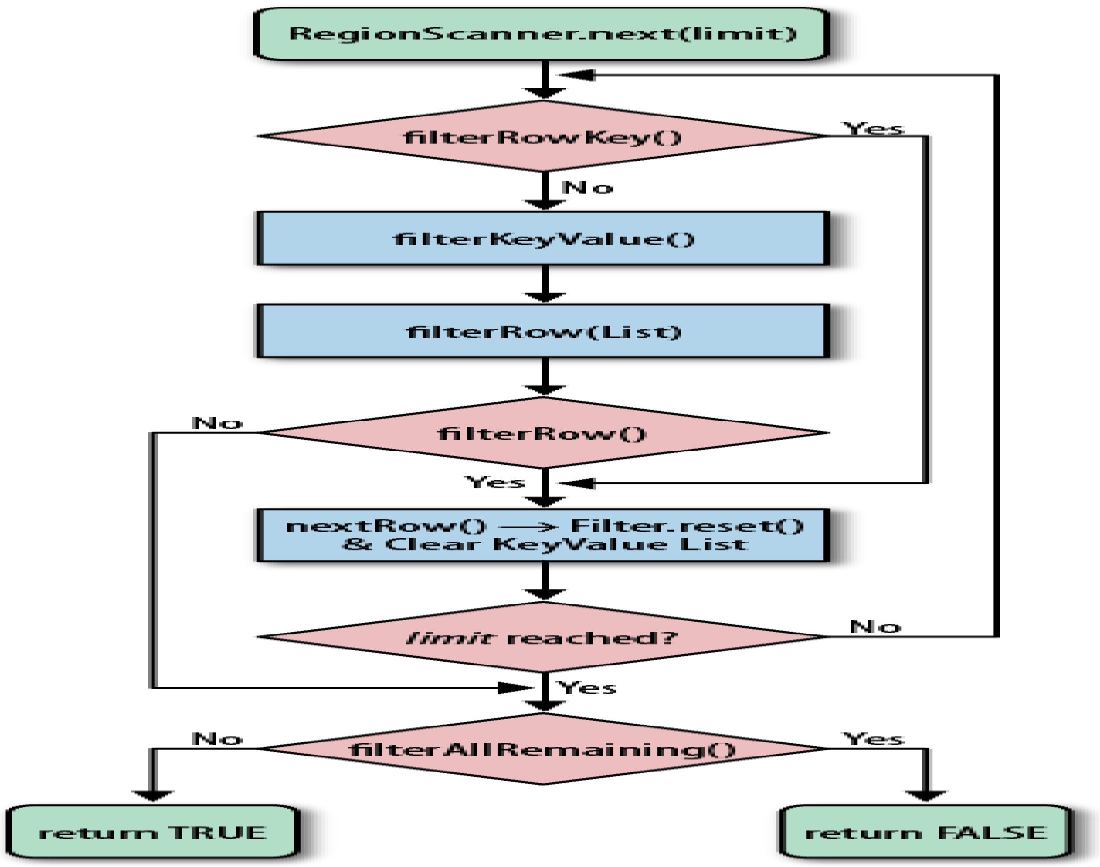
Comparator	**Description**
BinaryComparator	Uses Bytes.compareTo() to compare the current with the provided value.
BinaryPrefixComparator	Similar to the above, but does a lefthand, prefix-based match usingBytes.compareTo().
NullComparator	Does not compare against an actual value but whether a given one is null, or notnull.
BitComparator	Performs a bitwise comparison, providing a BitwiseOp class with AND,OR, and XOR operators.
RegexStringComparator	Given a regular expression at instantiation this comparator does a pattern match on the table data.
SubstringComparator	Treats the value and table data as Stringinstances

Comparator	Description
	and performs a contains()check.

FILTERROW() AND BATCH MODE

A filter using filterRow() to filter out an entire row, or filterRow(List) to modify the final list of included values, *must* also override the hasRowFilter() function to return true.

Figure shows the logical flow of the filter methods for a single row. There is a more fine-grained process to apply the filters on a column level, which is not relevant in this context.



Advanced Usage of HBASE:

It is important to have a good understanding of how to design tables, row keys, column names, and so on, to take full advantage of the architecture.

Key Design

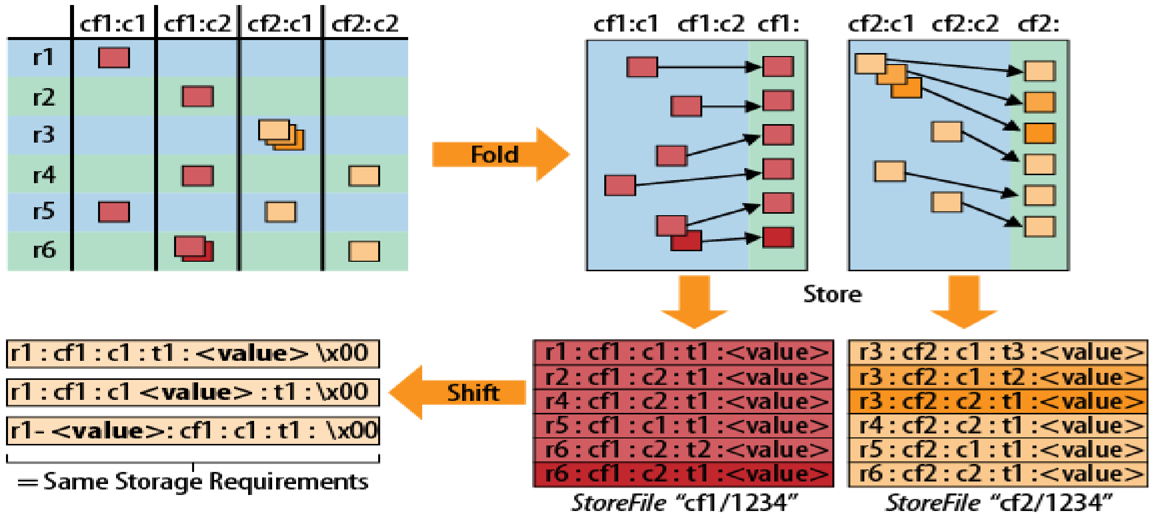
HBase has two fundamental *key* structures: the *row key* and the *column key*. Both can be used to convey meaning, by either the data they store, or by exploiting their sorting order. In the following sections, we will use these keys to solve commonly found problems when designing storage solutions.

Concepts

The first concept to explain in more detail is the logical layout of a table, compared to on-disk storage. HBase's main unit of separation within a table is the *column family*—not the actual columns as expected from a column-oriented database in their traditional sense.

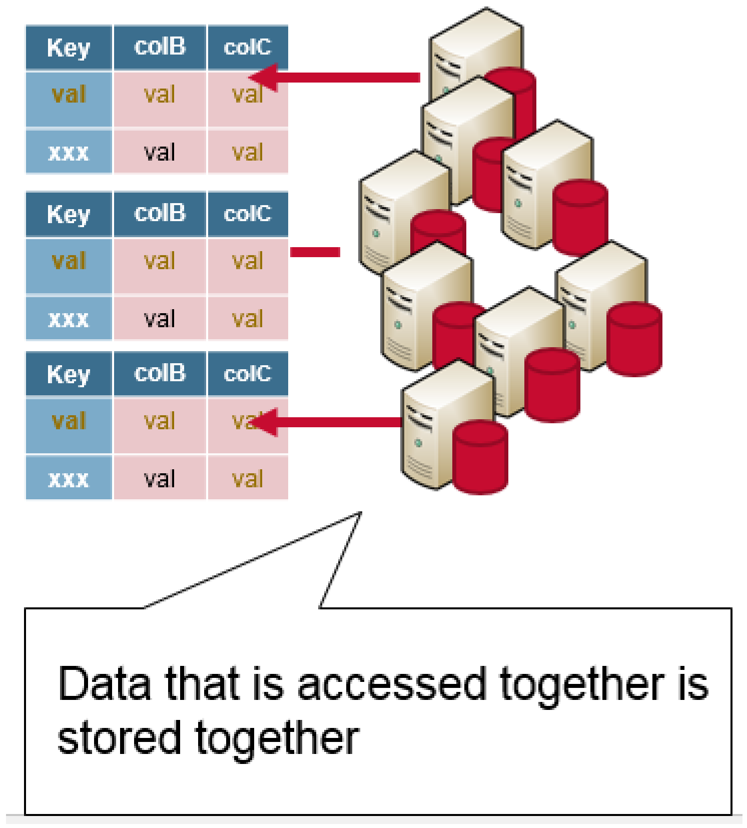
Figure - shows the fact that, although you store cells in a table format logically, in reality these rows are stored as linear sets of the actual cells, which in turn contain all the vital information inside them.

The top-left part of the figure shows the logical layout of your data—you have rows and columns. The columns are the typical HBase combination of a column family name and a column qualifier, forming the *column key*. The rows also have a *row key* so that you can address all columns in one logical row.


cf1:c1 cf1:c2 cf2:c1 cf2:c2
r1
r2
r3
r4
r5
r6
Fold
cf1:c1 cf1:c2 cf1:
cf2:c1 cf2:c2 cf2:
Store
r1 : cf1 : c1 : t1 : <value>
r2 : cf1 : c2 : t1 : <value>
r4 : cf1 : c2 : t1 : <value>
r5 : cf1 : c1 : t1 : <value>
r6 : cf1 : c2 : t2 : <value>
r6 : cf1 : c2 : t1 : <value>
StoreFile "cf1/1234"
r3 : cf2 : c1 : t3 : <value>
r3 : cf2 : c1 : t2 : <value>
r3 : cf2 : c2 : t1 : <value>
r4 : cf2 : c2 : t1 : <value>
r5 : cf2 : c1 : t1 : <value>
r6 : cf2 : c2 : t1 : <value>
StoreFile "cf2/1234"
Shift
r1 : cf1 : c1 : t1 : <value> \x00
r1 : cf1 : c1 <value> : t1 : \x00
r1- <value>: cf1 : c1 : t1 : \x00
= Same Storage Requirements

HBASE Design Schema

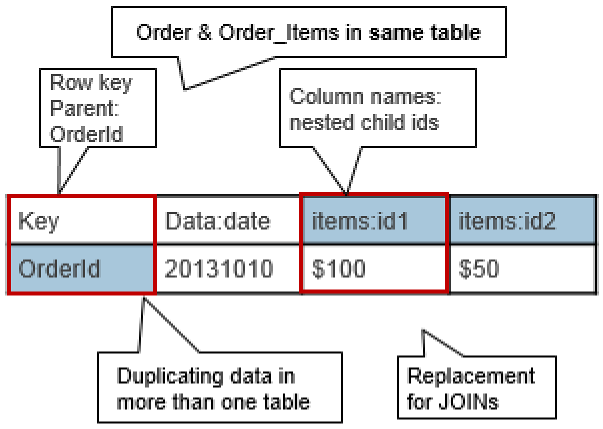
With HBase, you have a "query-first" schema design; all possible queries should be identified first, and the schema model designed accordingly. You should design your HBase schema to take advantage of the strengths of HBase. Think about your access patterns, and design your schema so that the data that is read together is stored together. Remember that HBase is designed for clustering.



- Distributed data is stored and accessed together
- It is query-centric, so focus on how the data is read
- Design for the questions

Parent-Child Relationship–Nested Entity

- Here is an example of denormalization in HBase, if your tables exist in a one-to-many relationship, it's possible to model it in HBase as a single row. In the example below, the order and related line items are stored together and can be read together with a get on the row key. This makes the reads a lot faster than joining tables together.


Order & Order_Items in same table
Row key Parent: OrderId
Column names: nested child ids
Key
Data:date
items:id1
items:id2
OrderId
20131010
$100
$50
Duplicating data in more than one table
Replacement for JOINs

- The rowkey corresponds to the parent entity id, the OrderId. There is one column family for the order data, and one column family for the order items. The Order Items are nested, the Order Item IDs are put into the column names and any non-identifying attributes are put into the value.

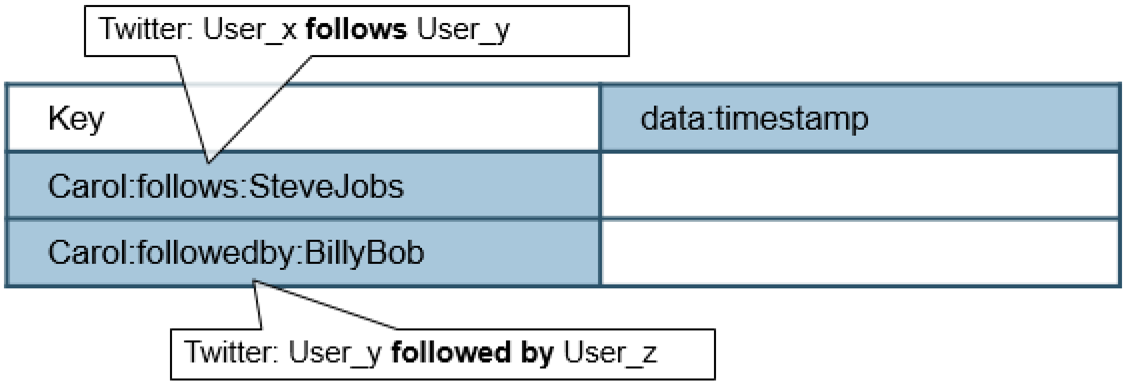
- This kind of schema design is appropriate when the only way you get at the child entities is via the parent entity.

Self-Join Relationship – HBase

- A self-join is a relationship in which both match fields are defined in the same table.

- Consider a schema for twitter relationships, where the queries are: which users does userX follow, and which users follow userX? Here’s a possible solution: The userids are put in a composite row key with the relationship type as a separator. For example, Carol follows Steve Jobs and Carol is followed by BillyBob. This allows for row key scans for everyone carol:follows or carol:followedby

- Below is the example Twitter table:



Key	data:timestamp
Carol:follows:SteveJobs	
Carol:followedby:BillyBob	

Twitter: User_x **follows** User_y
Twitter: User_y **followed by** User_z

Schema Design Exploration:

- Raw data from HDFS or HBase
- MapReduce for data transformation and ETL from raw data.
- Use bulk import from MapReduce to HBase
- Serve data for online reads from HBase

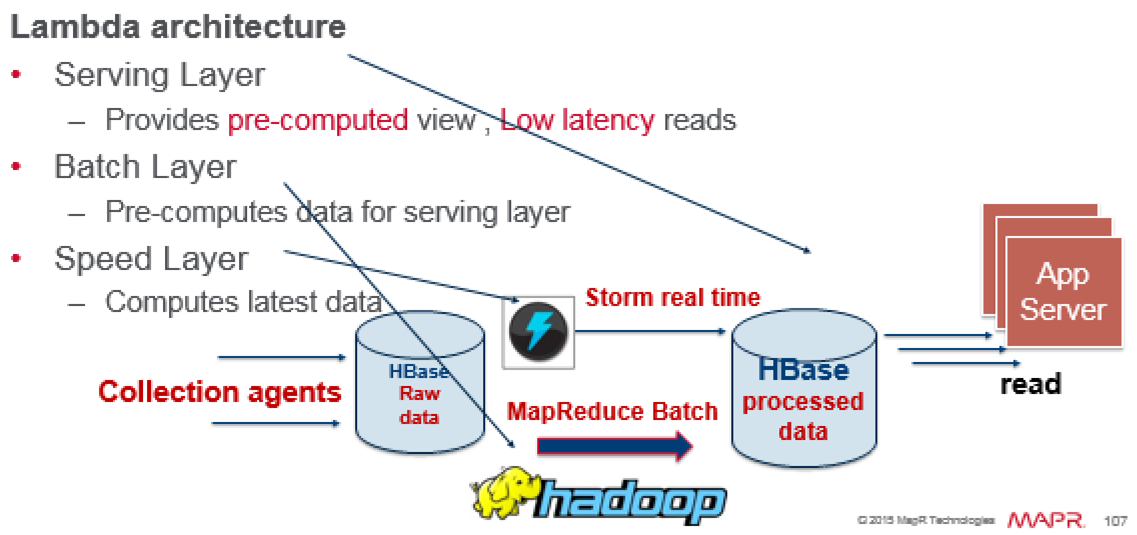
Designing for reads means aggressively de-normalizing data so that the data that is read together is stored together.

Data Access Pattern

The batch layer precomputes the batch views. In the batch view, you read the results from a precomputed view. The precomputed view is indexed so that it can be accessed quickly with random reads.

The serving layer indexes the batch view and loads it up so it can be efficiently queried to get particular values out of the view. A serving layer database only requires batch updates and random reads. The serving layer updates whenever the batch layer finishes precomputing a batch view.

You can do stream-based processing with Storm and batch processing with Hadoop. The speed layer only produces views on recent data, and is for functions computed on data in the few hours not covered by the batch.



Lambda architecture

- Serving Layer
  - Provides pre-computed view , Low latency reads
- Batch Layer
  - Pre-computes data for serving layer
- Speed Layer
  - Computes latest data

Advance Indexing In HBase

In HBase, the row key provides the same data retrieval benefits as a primary index. So, when you create a secondary index, use elements that are different from the row key.

Secondary indexes allow you to have a secondary way to read an HBase table. They provide a way to efficiently access records by means of some piece of information other than the primary key.

Secondary indexes require additional cluster space and processing because the act of creating a secondary index requires both space and processing cycles to update.

A method of index maintenance, called Diff-Index, can help IBM® Big SQL to create secondary indexes for HBase, maintain those indexes, and use indexes to speed up queries.

Why is this important?

With secondary indexing, I can either find a single row to find all of the rows that contain an attribute with a specific value.

Like everything in HBase, its stored in sort order so it becomes rather trivial for the client to fetch several rows and join them in sort order, or to take the intersection if we are trying to find the records that meet a specific qualification. (e.g. find all of Bob's employees who live in Cleveland, OH. Or find the average cost of repairing a Volvo S80 that was involved in a front end collision....)

As more people want to use HBase like a database and apply SQL, using secondary indexing makes filtering and doing data joins much more efficient. One just takes the intersection of the indexed qualifiers specified, and then apply the unindexed qualifiers as filters further reducing the resulset

Problems in Indexing

This design appears to mimic the concept of the column families, where like data is stored in a column family file. So that like data can be accessed quickly. But like the column families, we run in to the same problem... too many column families can be a bad thing when it comes to compaction.

Depending on the number of columns to be indexed, the size of the index table could easily be larger than the base table. (Especially when you add in geo-spatial indexing.)

Co-Processor

The idea of HBase Coprocessors was inspired by Google’s BigTable coprocessors. which Google developed to bring computing parallelism to BigTable. They have the following characteristics:

- Arbitrary code can run at each tablet in table server
- High-level call interface for clients

    - Calls are addressed to rows or ranges of rows and the coprocessor client library resolves them to actual locations;
    - Calls across multiple rows are automatically split into multiple parallelized RPC
- Provides a very flexible model for building distributed services
- Automatic scaling, load balancing, request routing for applications

Back to HBase, we definitely want to support efficient computational parallelism as well, beyond what Hadoop MapReduce can provide.

In addition, exciting new features can be built on top of it, for example secondary indexing, complex filtering (push down predicates), and access control.

Coprocessors can be loaded globally on all tables and regions hosted by the region server, these are known as system coprocessors; or the administrator can specify which coprocessors should be loaded on all regions for a table on a per-table basis, these are known as table coprocessors.

In order to support sufficient flexibility for potential coprocessor behaviors, two different aspects of extension are provided by the framework. One is the observer, which are like triggers in conventional databases, and the other is the endpoint, dynamic RPC endpoints that resemble stored procedures.

Observers

The idea behind observers is that we can insert user code by overriding upcall methods provided by the coprocessor framework. The callback functions are executed from core HBase code when certain events occur.

The coprocessor framework handles all of the details of invoking callbacks during various base HBase activities; the coprocessor need only insert the desired additional or alternate functionality.

The RegionObserver interface provides callbacks for:

- preOpen, postOpen: Called before and after the region is reported as online to the master.

- preFlush, postFlush: Called before and after the memstore is flushed into a new store file.
- preGet, postGet: Called before and after a client makes a Get request.
- preExists, postExists: Called before and after the client tests for existence using a Get.
- prePut and postPut: Called before and after the client stores a value.
- preDelete and postDelete: Called before and after the client deletes a value etc.

Endpoint

- As mentioned previously, observers can be thought of like database triggers. Endpoints, on the other hand, are more powerful, resembling stored procedures. One can invoke an endpoint at any time from the client.
- The endpoint implementation will then be executed remotely at the target region or regions, and results from those executions will be returned to the client.

- Endpoint is an interface for dynamic RPC extension. The endpoint implementation is installed on the server side and can then be invoked with HBase RPC. The client library provides convenience methods for invoking such dynamic interfaces.

In order to build and use your own endpoint, you need to:

- Have a new protocol interface which extends CoprocessorProtocol.
- Implement the Endpoint interface. The implementation will be loaded into and executed from the region context.
- Extend the abstract class BaseEndpointCoprocessor. This convenience class hides some internal details that the implementer need not necessary be concerned about, such as coprocessor framework class loading.
- On the client side, the Endpoint can be invoked by two new HBase client APIs:

    Executing against a single region:

```
o  HTableInterface.coprocessorProxy(Class<T> protocol, byte[] row)
```

Executing over a range of regions

```
o  HTableInterface.coprocessorExec(Class<T>     protocol,     byte[]
   startKey, byte[] endKey, Batch.Call<T,R> callable)
```

Coprocessor Management

After you have a good understanding of how coprocessors work in HBase, you can start to build your own experimental coprocessors, deploy them to your HBase cluster, and observe the new behaviors.

Build Your Own Coprocessor

We now assume you have your coprocessor code ready, compiled and packaged as a jar file.

Coprocessor Deployment

Currently we provide two options for deploying coprocessor extensions: load from configuration, which happens when the master or region servers start up; or load from table attribute, dynamic loading when the table is (re)opened. Because most users will set table attributes by way of the 'alter' command of the HBase shell, let's call this load from shell.

Load from Configuration

When a region is opened, the framework tries to read coprocessor class names supplied as the configuration entries:

- hbase.coprocessor.region.classes: for RegionObservers and Endpoints
- hbase.coprocessor.master.classes: for MasterObservers
- hbase.coprocessor.wal.classes: for WALObservers

Hers is an example of the hbase-site.xml where one RegionObserver is configured for all the HBase tables:

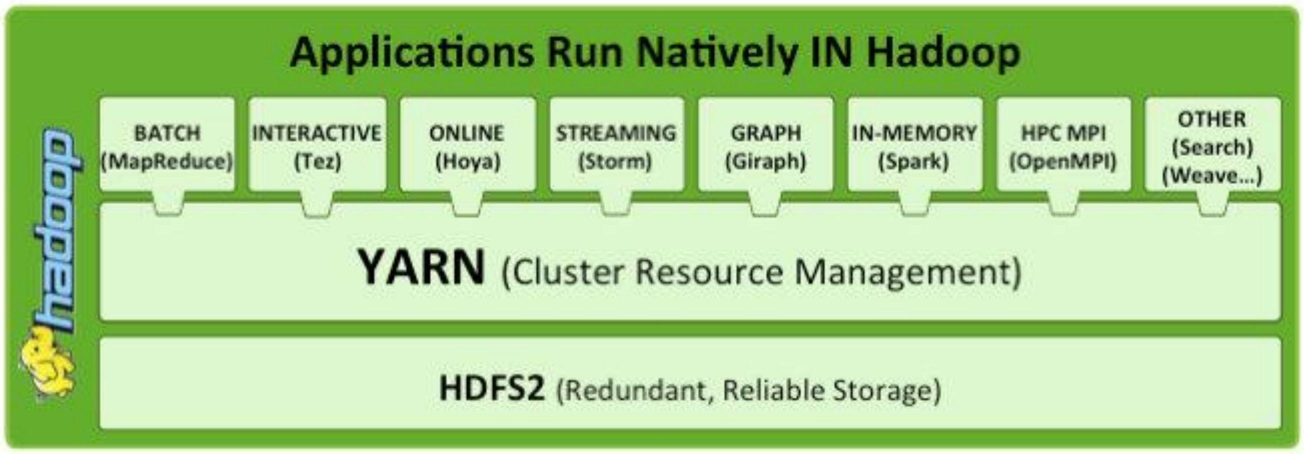
```
<property>
    <name>hbase.coprocessor.region.classes</name>
```

```
<value>org.apache.hadoop.hbase.coprocessor.AggregateImplementation</value>
 </property>
```

Hadoop 2.0

Apache Hadoop 2.0 represents a generational shift in the architecture of Apache Hadoop. With YARN, Apache Hadoop is recast as a significantly more powerful platform – one that takes Hadoop beyond merely batch applications to taking its position as a 'data operating system' where HDFS is the file system and YARN is the operating system.

YARN is a re-architecture of Hadoop that allows multiple applications to run on the same platform. With YARN, applications run "in" Hadoop, instead of "on" Hadoop:


Applications Run Natively IN Hadoop
BATCH (MapReduce)
INTERACTIVE (Tez)
ONLINE (Hoya)
STREAMING (Storm)
GRAPH (Giraph)
IN-MEMORY (Spark)
HPC MPI (OpenMPI)
OTHER (Search) (Weave...)
YARN (Cluster Resource Management)
HDFS2 (Redundant, Reliable Storage)
hadoop

The fundamental idea of YARN is to split up the two major responsibilities of the JobTracker and TaskTracker into separate entities. In Hadoop 2.0, the JobTracker and TaskTracker no longer exist and have been replaced by three components:

- **ResourceManager:** a scheduler that allocates available resources in the cluster amongst the competing applications.
- **NodeManager:** runs on each node in the cluster and takes direction from the ResourceManager. It is responsible for managing resources available on a single node.
- **ApplicationMaster:** an instance of a framework-specific library, an ApplicationMaster runs a specific YARN job and is responsible for negotiating resources from the

ResourceManager and also working with the NodeManager to execute and monitor Containers.

The actual data processing occurs within the Containers executed by the ApplicationMaster. A Container grants rights to an application to use a specific amount of resources (memory, cpu etc.) on a specific host.

YARN is not the only new major feature of Hadoop 2.0. HDFS has undergone a major transformation with a collection of new features that include:

- **NameNode HA:** automated failover with a hot standby and resiliency for the NameNode master service.
- **Snapshots:** point-in-time recovery for backup, disaster recovery and protection against use errors.
- **Federation:** a clear separation of namespace and storage by enabling generic block storage layer.

MRv2-YARN

The new architecture introduced in hadoop-0.23, divides the two major functions of the JobTracker: resource management and job life-cycle management into separate components.

The new ResourceManager manages the global assignment of compute resources to applications and the per-application ApplicationMaster manages the application’s scheduling and coordination.

An application is either a single job in the sense of classic MapReduce jobs or a DAG of such jobs.

The ResourceManager and per-machine NodeManager daemon, which manages the user processes on that machine, form the computation fabric.

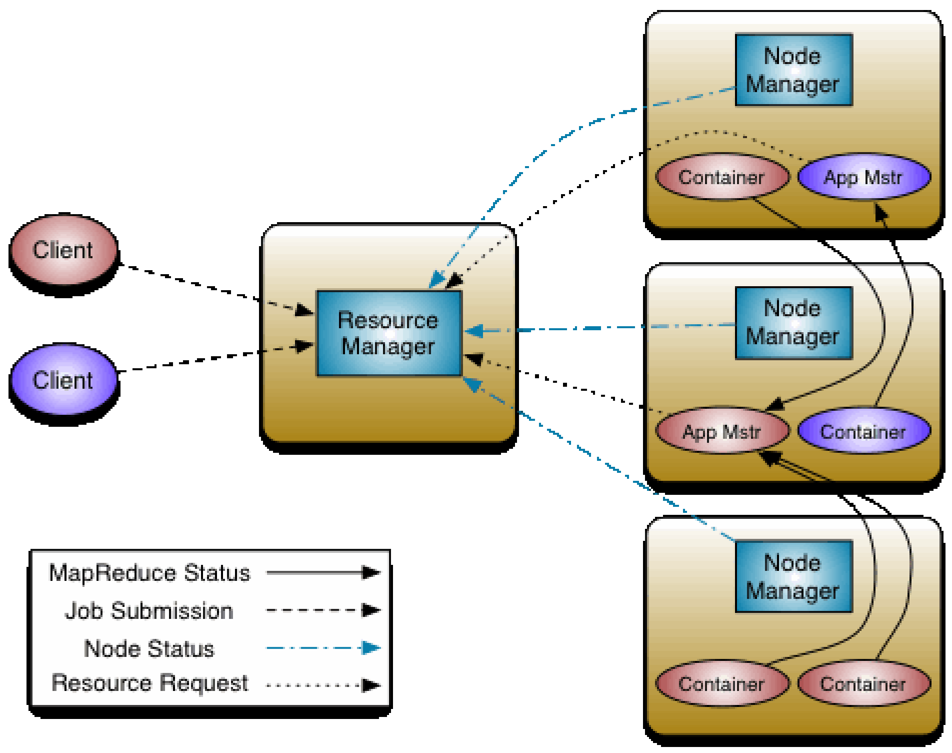
The per-application ApplicationMaster is, in effect, a framework specific library and is tasked with negotiating resources from the ResourceManager and working with the NodeManager(s) to execute and monitor the tasks.

he fundamental idea of YARN is to split up the functionalities of resource management and job scheduling/monitoring into separate daemons. The idea is to have a global ResourceManager (*RM*) and per-application ApplicationMaster (*AM*). An application is either a single job or a DAG of jobs.

The ResourceManager and the NodeManager form the data-computation framework. The ResourceManager is the ultimate authority that arbitrates resources among all the applications in the system.

The NodeManager is the per-machine framework agent who is responsible for containers, monitoring their resource usage (cpu, memory, disk, network) and reporting the same to the ResourceManager/Scheduler.

The per-application ApplicationMaster is, in effect, a framework specific library and is tasked with negotiating resources from the ResourceManager and working with the NodeManager(s) to execute and monitor the tasks.



The ResourceManager has two main components: Scheduler and ApplicationsManager.

The Scheduler is responsible for allocating resources to the various running applications subject to familiar constraints of capacities, queues etc. The Scheduler is pure scheduler in the sense that it performs no monitoring or tracking of status for the application. Also, it offers no guarantees about restarting failed tasks either due to application failure or hardware failures.

The Scheduler performs its scheduling function based the resource requirements of the applications; it does so based on the abstract notion of a resource *Container* which incorporates elements such as memory, cpu, disk, network etc.

NameNode High Availablity

Prior to Hadoop 2.0.0, the NameNode was a single point of failure (SPOF) in an HDFS cluster. Each cluster had a single NameNode, and if that machine or process became unavailable, the cluster as a whole would be unavailable until the NameNode was either restarted or brought up on a separate machine.

This impacted the total availability of the HDFS cluster in two major ways:

- In the case of an unplanned event such as a machine crash, the cluster would be unavailable until an operator restarted the NameNode.
- Planned maintenance events such as software or hardware upgrades on the NameNode machine would result in windows of cluster downtime.

The HDFS High Availability feature addresses the above problems by providing the option of running two redundant NameNodes in the same cluster in an Active/Passive configuration with a hot standby.

This allows a fast failover to a new NameNode in the case that a machine crashes, or a graceful administrator-initiated failover for the purpose of planned maintenance.

Architecture

In a typical HA cluster, two separate machines are configured as NameNodes. At any point in time, exactly one of the NameNodes is in an *Active* state, and the other is in a *Standby* state. The Active NameNode is responsible for all client operations in the cluster, while the Standby is simply acting as a slave, maintaining enough state to provide a fast failover if necessary.

In order for the Standby node to keep its state synchronized with the Active node, the current implementation requires that the two nodes both have access to a directory on a shared storage device (eg an NFS mount from a NAS). This restriction will likely be relaxed in future versions.

When any namespace modification is performed by the Active node, it durably logs a record of the modification to an edit log file stored in the shared directory. The Standby node is constantly watching this directory for edits, and as it sees the edits, it applies them to its own namespace.

In the event of a failover, the Standby will ensure that it has read all of the edits from the shared storage before promoting itself to the Active state. This ensures that the namespace state is fully synchronized before a failover occurs.

In order to provide a fast failover, it is also necessary that the Standby node have up-to-date information regarding the location of blocks in the cluster. In order to achieve this, the DataNodes are configured with the location of both NameNodes, and send block location information and heartbeats to both.

Hardware resources

In order to deploy an HA cluster, you should prepare the following:

- **NameNode machines** - the machines on which you run the Active and Standby NameNodes should have equivalent hardware to each other, and equivalent hardware to what would be used in a non-HA cluster.
- **Shared storage** - you will need to have a shared directory which both NameNode machines can have read/write access to. Typically this is a remote filer which supports

NFS and is mounted on each of the NameNode machines. Currently only a single shared edits directory is supported.

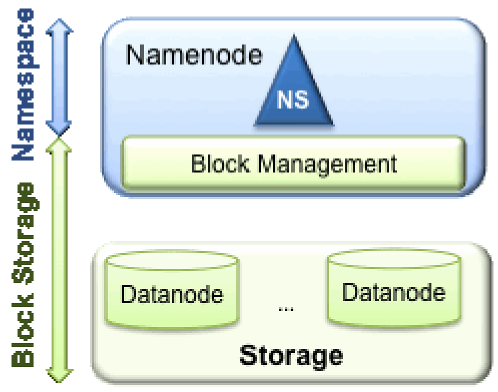
Thus, the availability of the system is limited by the availability of this shared edits directory, and therefore in order to remove all single points of failure there needs to be redundancy for the shared edits directory.

HDFS Federation

HDFS Federation improves the existing **HDFS** architecture through a clear separation of namespace and storage, enabling generic block storage layer. It enables support for multiple namespaces in the cluster to improve scalability and isolation

This guide provides an overview of the HDFS Federation feature and how to configure and manage the federated cluster.

Background



HDFS has two main layers:

- **Namespace**
    - Consists of directories, files and blocks.
    - It supports all the namespace related file system operations such as create, delete, modify and list files and directories.
- **Block Storage Service**, which has two parts:
    - Block Management (performed in the Namenode)

        - Provides Datanode cluster membership by handling registrations, and periodic heart beats.
        - Processes block reports and maintains location of blocks.
        - Supports block related operations such as create, delete, modify and get block location.
        - Manages replica placement, block replication for under replicated blocks, and deletes blocks that are over replicated.
    - Storage - is provided by Datanodes by storing blocks on the local file system and allowing read/write access.

The prior HDFS architecture allows only a single namespace for the entire cluster. In that configuration, a single Namenode manages the namespace. HDFS Federation addresses this limitation by adding support for multiple Namenodes/namespaces to HDFS.

Key Benefits

- Namespace Scalability - Federation adds namespace horizontal scaling. Large deployments or deployments using lot of small files benefit from namespace scaling by allowing more Namenodes to be added to the cluster.
- Performance - File system throughput is not limited by a single Namenode. Adding more Namenodes to the cluster scales the file system read/write throughput.
- Isolation - A single Namenode offers no isolation in a multi user environment. For example, an experimental application can overload the Namenode and slow down production critical applications. By using multiple Namenodes, different categories of applications and users can be isolated to different namespaces.

Federation configuration is **backward compatible** and allows existing single Namenode configurations to work without any change. The new configuration is designed such that all the nodes in the cluster have the same configuration without the need for deploying different configurations based on the type of the node in the cluster.

Configuration:

Step 1: Add the dfs.nameservices parameter to your configuration and configure it with a list of comma separated NameServiceIDs. This will be used by the Datanodes to determine the Namenodes in the cluster.

Step 2: For each Namenode and Secondary Namenode/BackupNode/Checkpointer add the following configuration parameters suffixed with the corresponding NameServiceID into the common configuration file:

Formatting Namenodes

Step 1: Format a Namenode using the following command:

```
[hdfs]$ $HADOOP_PREFIX/bin/hdfs namenode -format [-clusterId <cluster_id>]
```

Choose a unique cluster_id which will not conflict other clusters in your environment. If a cluster_id is not provided, then a unique one is auto generated.

Step 2: Format additional Namenodes using the following command:

```
[hdfs]$ $HADOOP_PREFIX/bin/hdfs namenode -format -clusterId <cluster_id>
```

Upgrading from an older release and configuring federation

Older releases only support a single Namenode. Upgrade the cluster to newer release in order to enable federation During upgrade you can provide a ClusterID as follows:

```
[hdfs]$ $HADOOP_PREFIX/bin/hdfs start namenode --config $HADOOP_CONF_DIR  -
upgrade -clusterId <cluster_ID>
```

If cluster_id is not provided, it is auto generated.

Adding a new Namenode to an existing HDFS cluster

Perform the following steps:

- Add dfs.nameservices to the configuration.

- Update the configuration with the NameServiceID suffix. Configuration key names changed post release 0.20. You must use the new configuration parameter names in order to use federation.
- Add the new Namenode related config to the configuration file.
- Propagate the configuration file to the all the nodes in the cluster.
- Start the new Namenode and Secondary/Backup.
- Refresh the Datanodes to pickup the newly added Namenode by running the following command against all the Datanodes in the cluster:

```
[hdfs]$ $HADOOP_PREFIX/bin/hdfs dfsadmin -refreshNameNodes
<datanode_host_name>:<datanode_rpc_port>
```

Managing the cluster

Starting and stopping cluster

To start the cluster run the following command:

```
[hdfs]$ $HADOOP_PREFIX/sbin/start-dfs.sh
```

To stop the cluster run the following command:

```
[hdfs]$ $HADOOP_PREFIX/sbin/stop-dfs.sh
```

Balancer

The Balancer has been changed to work with multiple Namenodes. The Balancer can be run using the command:

```
[hdfs]$ $HADOOP_PREFIX/sbin/hadoop-daemon.sh start balancer [-policy <policy>]
```

Decommissioning

Decommissioning is similar to prior releases. The nodes that need to be decomissioned are added to the exclude file at all of the Namenodes. Each Namenode decommissions its Block Pool. When all the Namenodes finish decommissioning a Datanode, the Datanode is considered decommissioned.

Step 1: To distribute an exclude file to all the Namenodes, use the following command:

```
[hdfs]$ $HADOOP_PREFIX/sbin/distribute-exclude.sh <exclude_file>
```

Step 2: Refresh all the Namenodes to pick up the new exclude file:

```
[hdfs]$ $HADOOP_PREFIX/sbin/refresh-namenodes.sh
```

The above command uses HDFS configuration to determine the configured Namenodes in the cluster and refreshes them to pick up the new exclude file.

Cluster Web Console

Similar to the Namenode status web page, when using federation a Cluster Web Console is available to monitor the federated cluster at http://<any_nn_host:port>/dfsclusterhealth.jsp. Any Namenode in the cluster can be used to access this web page.

Migrating from MapReduce 1 (MRv1) to MapReduce 2

MapReduce from Hadoop 1 (MapReduce MRv1) has been split into two components. The cluster resource management capabilities have become YARN (Yet Another Resource Negotiator), while the MapReduce-specific capabilities remain MapReduce.

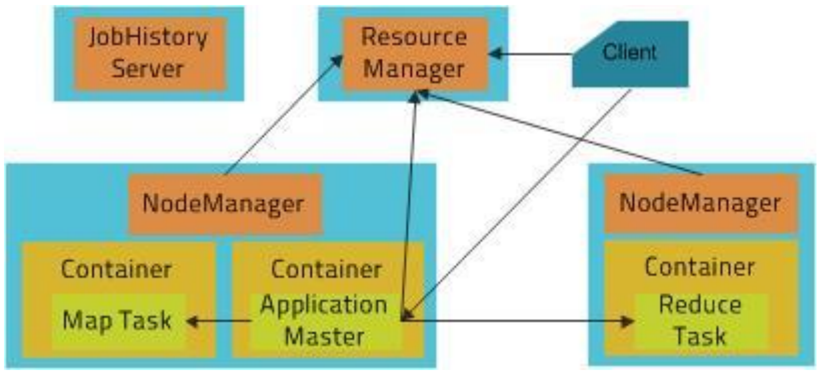
In the MapReduce MRv1 architecture, the cluster was managed by a service called the JobTracker. TaskTracker services lived on each host and would launch tasks on behalf of jobs. The JobTracker would serve information about completed jobs.

In MapReduce MRv2, the functions of the JobTracker have been split between three services. The ResourceManager is a persistent YARN service that receives and runs applications (a MapReduce job is an application) on the cluster.

It contains the scheduler, which, as previously, is pluggable. The MapReduce-specific capabilities of the JobTracker have been moved into the MapReduce Application Master, one of which is started to manage each MapReduce job and terminated when the job completes.

The JobTracker function of serving information about completed jobs has been moved to the JobHistory Server. The TaskTracker has been replaced with the NodeManager, a YARN service that manages resources and deployment on a host. It is responsible for launching containers, each of which can house a map or reduce task.

Nearly all jobs written for MRv1 will be able to run without any modifications on an MRv2 cluster.



The new architecture has its advantages. First, by breaking up the JobTracker into a few different services, it avoids many of the scaling issues faced by MapReduce in Hadoop 1.

More importantly, it makes it possible to run frameworks other than MapReduce on a Hadoop cluster. For example, Impala can also run on YARN and share resources with MapReduce.

Configuration Migration

Since MapReduce 1 functionality has been split into two components, MapReduce cluster configuration options have been split into YARN configuration options, which go in yarn-site.xml, and MapReduce configuration options, which go in mapred-site.xml. Many have been given new names to reflect the shift. As JobTrackers and TaskTrackers no longer exist in MRv2, all configuration options pertaining to them no longer exist, although many have corresponding options for the ResourceManager, NodeManager, and JobHistoryServer.

A minimal configuration required to run MRv2 jobs on YARN is:

yarn-site.xml configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
```

```
        <property>
          <name>yarn.resourcemanager.hostname</name>
          <value>you.hostname.com</value>
        </property>

        <property>
          <name>yarn.nodemanager.aux-services</name>
          <value>mapreduce_shuffle</value>
        </property>
</configuration>
```

mapred-site.xml configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <property>
    <name>mapreduce.framework.name</name>
    <value>yarn</value>
  </property>
</configuration>
```

Below is a table with HA-related configurations used in MRv1 and their equivalents in YARN:

MRv1	**YARN / MRv2**	**Comment**
mapred.jobtrackers.*name*	yarn.resourcemanager.ha.rm-ids	
mapred.ha.jobtracker.id	yarn.resourcemanager.ha.id	Unlike in MRv1, this must be configured in YARN.
mapred.jobtracker.*rpc-address.name.id*	yarn.resourcemanager.*rpc-address.id*	YARN/ MRv2 has different RPC ports for different functionalities. Each port-related configuration must be suffixed with an id. Note that there is no <name> in YARN.
mapred.ha.jobtracker.*rpc-address.name.id*	yarn.resourcemanager.ha.admin.address	
mapred.ha.fencing.methods	yarn.resourcemanager.ha.fencer	Not required to be specified
mapred.client.failover.*	None	Not required
	yarn.resourcemanager.ha.enabled	Enable HA
mapred.jobtracker.restart.recover	yarn.resourcemanager.recovery.enabled	Enable recovery of jobs after failover
	yarn.resourcemanager.store.class	org.apache .hadoop.yarn

MRv1	YARN / MRv2	Comment
		.server.resourcemanager .recovery .ZKRMStateStore
mapred.ha.automatic-failover.enabled	yarn.resourcemanager.ha.automatic-failover.enabled	Enable automatic failover
mapred.ha.zkfc.port	yarn.resourcemanager.ha.automatic-failover.port	
mapred.job.tracker	yarn.resourcemanager.cluster.i	

Programming in YARN Framework

Under Hadoop 2.0, MapReduce is but one instance of a YARN application, where YARN has taken center stage as the "operating system" of Hadoop. Because YARN allows *any* application to run on equal footing with MapReduce, it opened the floodgates for a new generation of software applications with these kinds of features:

More programming models. Because YARN supports any application that can divide itself into parallel tasks, they are no longer shoehorned into the palette of "mappers," "combiners," and "reducers." This in turn supports complex data-flow applications like ETL and ELT, and iterative programs like massively-parallel machine learning and modeling.

Integration of native libraries. Because YARN has robust support for *any* executable – not limited to MapReduce, and not even limited to Java – application vendors with a large mature code base have a clear path to Hadoop integration.

Support for large reference data. YARN automatically "localizes" and caches large reference datasets, making them available to all nodes for "data local" processing. This supports legacy functions like address standardization, which require large reference data sets that cannot be accessed from the Hadoop Distributed File System (HDFS) by the legacy libraries.

Of course, MapReduce isn't the only option for processing data at scale using Hadoop. Tools like Pig (a large scale query and analysis system), Hive (a data warehousing application) and others have been available for some time. These tools can express transforms and analysis using more accessible constructs: Hive uses HQL, a language similar to SQL.

Pig provides a script language (Pig Latin) to create MapReduce jobs. Business analysts familiar with conventional tools like SQL and SAS should be able to use these tools to write programs to solve large data problems on Hadoop clusters

The Value of Visual Application Development Tools

A new generation of "visual design" application development tools could help solve these coding problems. By running as native YARN applications and side-stepping the need for MapReduce, some of these programs eliminate coding altogether.

Other tools reduce coding by generating MapReduce code or by generating scripts like Pig. Visual designers are powerful for several reasons:

• Increased level of abstraction: Instead of thinking about classes and methods, users see operations, data, and outcomes.

• Fast "what-if": The drag-and-connect interface supports quick try/observe/adjust cycles.

• Automatic optimization: Scaling and efficiency are built-in.

• High-level palette: High-level constructs like "standardize address", "deduplicate consumers", or "parse names" are often directly on the designer palette.

How does this look in practice? Here's an illustration that shows how three competing approaches differ:

• MapReduce written in Java

• Pig scripts developed from scratch

• A visually-designed process running a native YARN ETL application. The application is from RedPoint Global, but comparable approaches can be seen in Talend and Actian.

Using these three approaches, we conducted a "Word Count" test on 30,000 files (20 gigabytes) of Project Gutenberg books. This test reads lines of text, breaks them into words, and creates a concordance (list of words and the number of times each occurs).

Our Hadoop cluster was small—only four nodes—but was large enough to demonstrate the concepts and tradeoffs

MapReduce:

Set-up time: While flexible, MapReduce had the longest learning curve and required significant coding skills—both as a Java programmer and a MapReduce specialist—to prepare the test.

Performance: It took 3 hours 20 minutes to run the test initially due to the “small files problem” that is familiar to seasoned MapReduce programmers.

This problem occurs when reading large collections of small files, because MapReduce’s default behavior is to assign a mapper task to each file. This results in a huge number of tasks.

To address this issue, we created a custom InputFormat class to read multiple files at once. This reduced our run time to 58 minutes. Then we tuned the split sizes and mapper task limit appropriately, which dropped the run time to about six minutes.

Comments: Each performance improvement came at a cost. Overall, nearly a full day of programmer time was spent optimizing the original code.

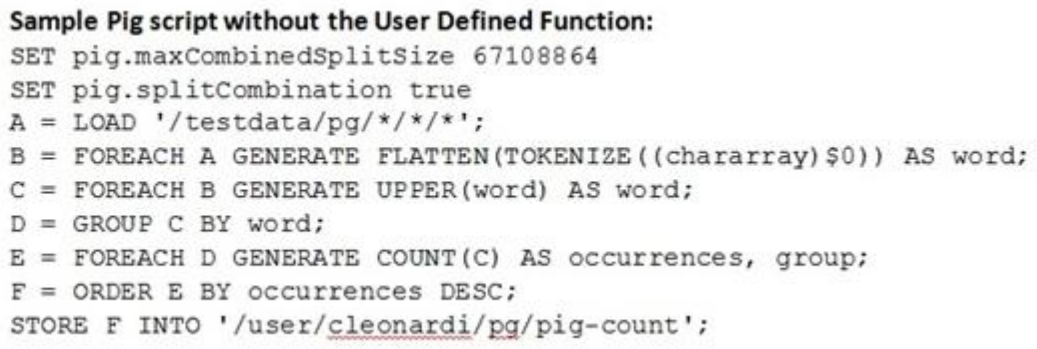
Pig:

Set-up time: Learning Pig was fairly easy. It was pretty natural to create the coding for this test. However to make a common adjustment in the code—changing the set of whitespace separators to include punctuation—required the addition of a “User defined function” or UDF which had to be written in Java.

Pig is generally easy enough to use by people who aren’t professional programmers but who know how to write scripting languages like JavaScript or Visual Basic.

Performance: The results were not stellar: run time was close to 15 minutes.

Comments: While coding took less time, Java programming was ultimately required to meet the test requirements.



Sample Pig script without the User Defined Function:

```
SET pig.maxCombinedSplitSize 67108864
SET pig.splitCombination true
A = LOAD '/testdata/pg/*/*/*';
B = FOREACH A GENERATE FLATTEN(TOKENIZE((chararray)$0)) AS word;
C = FOREACH B GENERATE UPPER(word) AS word;
D = GROUP C BY word;
E = FOREACH D GENERATE COUNT(C) AS occurrences, group;
F = ORDER E BY occurrences DESC;
STORE F INTO '/user/cleonardi/pg/pig-count';
```

YARN-enabled ETL/ELT designer:

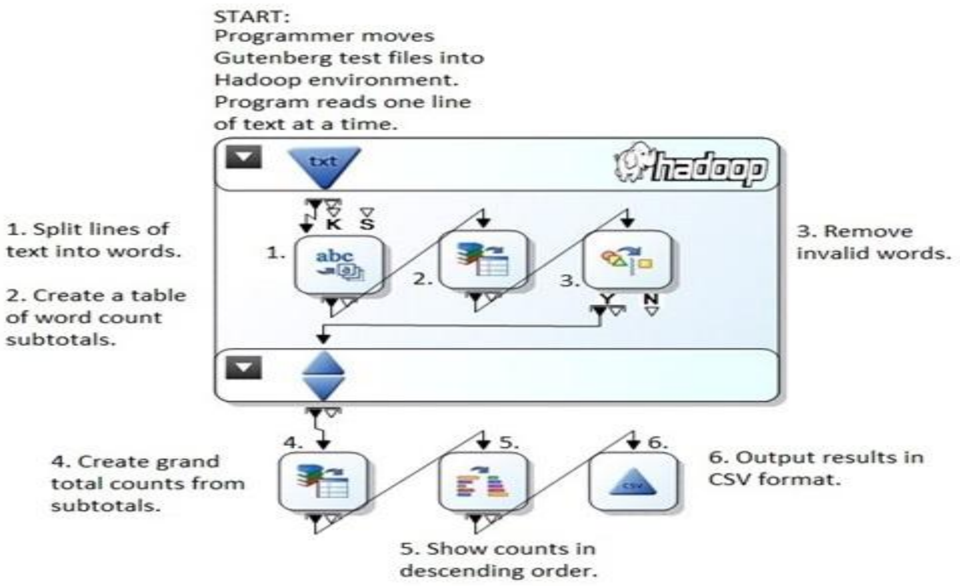
Set-up time: The tool is designed to have a shorter learning curve than even Pig scripting. Dragging tools like "Delimited Input", "Summarize" and "Tokenize" from the palette and configuring them is designed to be discoverable and intuitive, and the resulting diagram has a one-to-one correspondence between icons and operations. There's no need for coding or learning a language like Java or Pig.

The visual design covers the input file format, tokenizing and counting steps. The resulting data flow graph contains seven icons along with a grouping construct that shows what executes "inside" Hadoop. Each icon represents a step in the data transformation.

Performance: The run time for this data flow is just over three minutes with no tuning.

Comments: Because there is no code to manage, and editing is done visually, running "what if" scenarios is quick for non-programmers.

Once the data flow is designed, it can be stored and saved for later use. In addition, the logic can be captured into a "macro" for sharing and reuse between multiple data flows.


START:
Programmer moves Gutenberg test files into Hadoop environment. Program reads one line of text at a time.
txt
hadoop
1. Split lines of text into words.
2. Create a table of word count subtotals.
3. Remove invalid words.
4. Create grand total counts from subtotals.
5. Show counts in descending order.
6. Output results in CSV format.
1.
2.
3.
4.
5.
6.
K
S
Y
N
abc
csv

<u>Oozie workflow scheduler for Hadoop</u>

Oozie is a framework that helps automate this process and codify this work into repeatable units or workflows that can be reused over time without the need to write any new code or steps.

Apache Oozie is an open source project based on Java™ technology that simplifies the process of creating workflows and managing coordination among jobs. In principle, Oozie offers the ability to combine multiple jobs sequentially into one logical unit of work.

One advantage of the Oozie framework is that it is fully integrated with the Apache Hadoop stack and supports Hadoop jobs for Apache MapReduce, Pig, Hive, and Sqoop.

. In addition, it can be used to schedule jobs specific to a system, such as Java programs.

In practice, there are different types of Oozie jobs:

- *Oozie Workflow* jobs — Represented as directed acyclical graphs to specify a sequence of actions to be executed.
- *Oozie Coordinator* jobs — Represent Oozie workflow jobs triggered by time and data availability.

Oozie Bundle— Facilitates packaging multiple coordinator and workflow jobs, and makes it easier to manage the life cycle of those jobs.

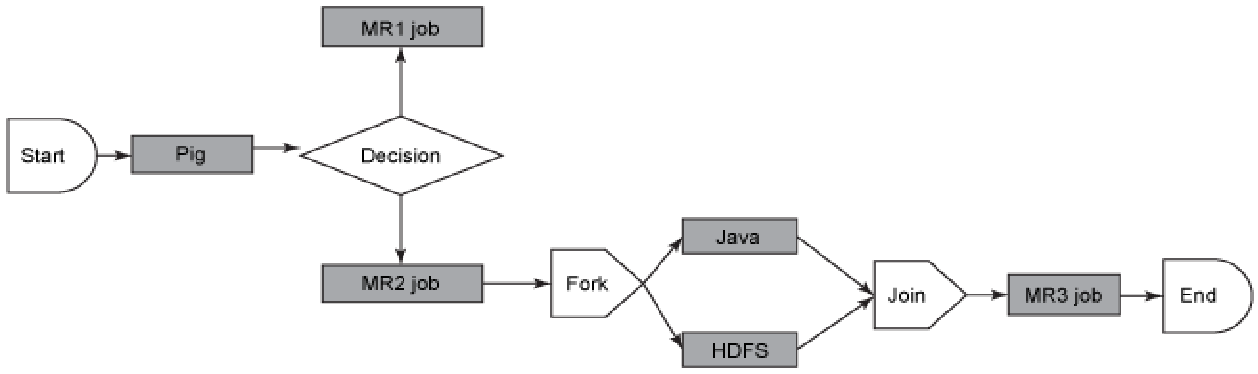
How does Oozie work?

An Oozie workflow is a collection of actions arranged in a directed acyclic graph (DAG). This graph can contain two types of nodes: control nodes and action nodes. *Control nodes*, which are used to define job chronology, provide the rules for beginning and ending a workflow and control the workflow execution path with possible decision points known as fork and join nodes.

Action nodes are used to trigger the execution of tasks. In particular, an action node can be a MapReduce job, a Pig application, a file system task, or a Java application. (The shell and ssh actions have been deprecated).

Oozie is a native Hadoop stack integration that supports all types of Hadoop jobs and is integrated with the Hadoop stack.

Figure shown below illustrates a sample Oozie workflow that combines six action nodes (Pig scrip, MapReduce jobs, Java code, and HDFS task) and five control nodes (Start, Decision control, Fork, Join, and End). Oozie workflows can be also parameterized. When submitting a workflow job, values for the parameters must be provided. If the appropriate parameters are used, several identical workflow jobs can occur concurrently.



In practice, it is sometimes necessary to run Oozie workflows on regular time intervals, but in coordination with other conditions, such as the availability of specific data or the completion of any other events or tasks.

Oozie in action

Use an Oozie workflow to run a recurring job. Oozie workflows are written as an XML file representing a directed acyclic graph. Let's look at the following simple workflow example that chains two MapReduce jobs. The first job performs an initial ingestion of the data and the second job merges data of a given type.

Simple example of Oozie workflow

```
<workflow-app xmlns='uri:oozie:workflow:0.1' name='SimpleWorkflow'>
        <start to='ingestor'/>
        <action name='ingestor'>
                </java>
                        <job-tracker>${jobTracker}</job-tracker>
                        <name-node>${nameNode}</name-node>
                        <configuration>
                                <property>
                                        <name>mapred.job.queue.name</name>
                                        <value>default</value>
                                </property>
                        </configuration>
                        <arg>${driveID}</arg>
                </java>
                <ok to='merging'/>
                <error to='fail'/>
        </action>
        <fork name='merging'>
                <path start='mergeT1'/>
                <path start='mergeT2'/>
        </fork>
        <action name='mergeT1'>
                <java>
                        <job-tracker>${jobTracker}</job-tracker>
                        <name-node>${nameNode}</name-node>
                        <configuration>
                                <property>
                                        <name>mapred.job.queue.name</name>
                                        <value>default</value>
                                </property>
                        </configuration>
                        <arg>-drive</arg>
                        <arg>${driveID}</arg>
                        <arg>-type</arg>
                        <arg>T1</arg>
                </java>
                <ok to='completed'/>
                <error to='fail'/>
        </action>
        <action name='mergeT2'>
                <java>
                        <job-tracker>${jobTracker}</job-tracker>
                        <name-node>${nameNode}</name-node>
                        <configuration>
                                <property>
                                        <name>mapred.job.queue.name</name>
                                        <value>default</value>
                                </property>
                        </configuration>
                        <main-
class>com.navteq.assetmgmt.hdfs.merge.MergerLoader</main-class>
                        <arg>-drive</arg>
                        <arg>${driveID}</arg>
                        <arg>-type</arg>
                        <arg>T2</arg>
                </java>
                <ok to='completed'/>
                <error to='fail'/>
        </action>
        <join name='completed' to='end'/>
        <kill name='fail'>
                <message>Java failed, error
message[${wf:errorMessage(wf:lastErrorNode())}]</message>
        </kill>
        <end name='end'/>
</workflow-app>
```