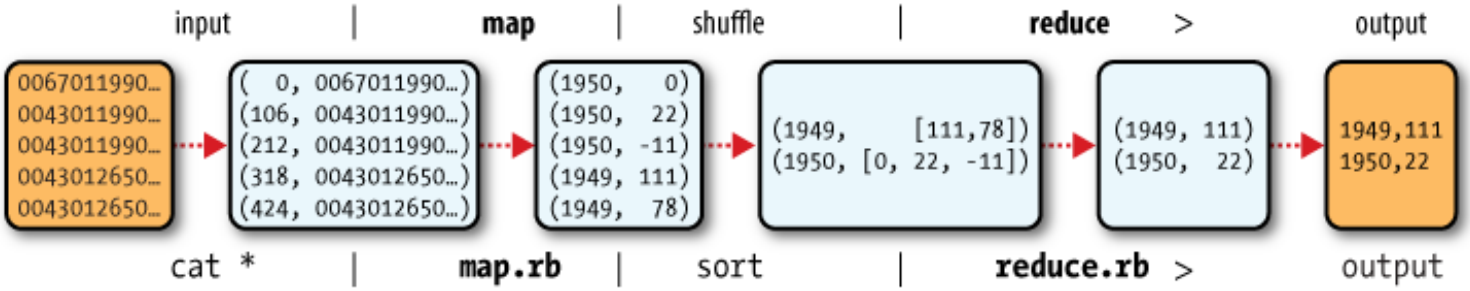
Mapreduce

Analyzing the Data with Hadoop MapReduce

Each phase has key-value pairs as input and output that may be chosen by the programmer. The input of the map phase is a text input that gives each line in a dataset as text value. The key is the offset from the beginning of the file so:

```
key                 value
[line-number]       [content of the line]
1                   First line of the dataset
2                   Second line...
50                  Line 50th of the dataset
```

The output of the Map is processed by the MapReduce framework before being sent to the Reduce:



Implementation: Map

The Map declares an abstract map() method and extends Mapper with four formal type parameters that, in order, are input and output key-value in that order. Also provides a Context object to write the output to. Hadoop uses its own implementations of Java types for IO, for example:

```
Long        -> LongWritable
String      -> Text
Integer     -> IntWritable
Null        -> NullWritable
Double      -> DoubleWritable
Object      -> ObjectWritable
Short       -> ShortWritable
Map         -> MapWritable
```

```
Tuple       -> TupleWritable
```

Implementation: Reduce

The Reducer declares the reduce() method and takes also 4 parameters, the first 2 must match the last 2 of the Mapper. It also gives a Context object to write.

Implementation: Job

In the job you match the Mapper and the Reducer. Generally this methods are critical:

- **Job.setJarByClass()** to specify the name of the Job class
- **Job.setJobName()** to give a name to the job
- **File[Input/Output]Format.add[Input/Output]Path()** To specify the input dataset and output result
- **Job.set[Mapper/Reducer/Combiner]Class()** to specify the classes of the mapper, reducer and combiner class
- **Job.setOutput[Key/Value]Class()** to specify the Hadoop classes of the output result
- **Job.waitForCompletion(true) ? 0 : 1** Submits the job and waits until it finish. The true indicates to output information to the console of the job and the 0/1 indicates the success or fail of the job.
- **Job.setCombiner

Running the Job

Run the jar with hadoop command as it has the source libraries (java does not).

Result

The output gives lots of info: INFO mapreduce.Job: Running job: job_local_0001 -> id of the job INFO mapred.LocalJobRunner: Starting task: attempt_local_0001_m_000000_0 -> map task id INFO mapred.Task: Task 'attempt_local_0001_m_000000_0' done -> map task finished INFO mapred.Task: Task 'attempt_local_0001_r_000000_0' done. -> reduce task finished INFO output.FileOutputCommitter: Saved output of task ... to file:/... -> output result and path

Scaling out

Data Flow

The jobtracker coordinates all the jobs run on the system by scheduling tasks to run on tasktrackers. Tasktrackers run tasks and send progress reports to the jobtracker, which keeps a record of the overall progress of each job.

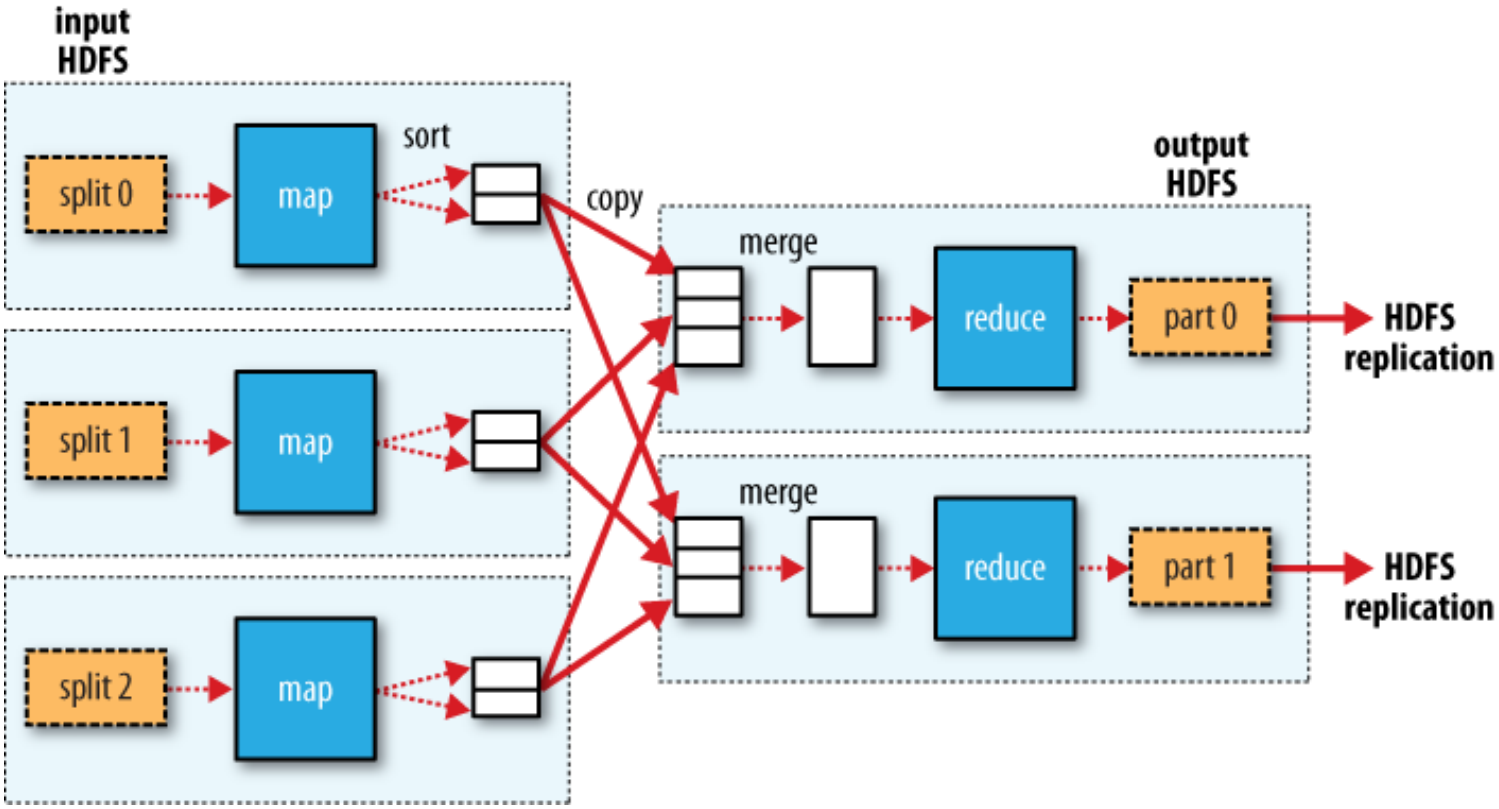
Hadoop divides inputs in **input-splits** or **splits** with one map for each split for each *record* in the split. Many small splits can be well processed in parallel but they can't be too small. 64 mb is the recommended split size.

Data locality optimization is when Hadoop runs jobs with data in its HDFS because don't use too much bandwith. Sometimes this is not possible as all nodes with replicas of the split are already running a Map. Then it will look for a free map slot in the same rack or outside the rack if there are not free nodes in the same rack.

With more than one-block, its difficult that the same node has more than one of the splits so some of the split must be transferred throught the nodes.

Map writes the output to the local disk, not to HDFS because it is an intermediate output before taking it to the reducer. To pass it to HDFS (with the replication) would be overkill.

Reduce don't have data locality optimization. The input to a single reduce is usually the output of all mappers that has benn network transferred. The first replica of the reduce is written in the local HDFS and other will be written in off-rack nodes.



They can be one or multiple reducers tasks. If there are more than one the mapper partition the output but

the records for a given key will be all in the same partition always.

Combiner

A combiner runs the reduce task at end of each mapper process before submitting to the reducer. This helps reduce data transferred between map and reduce. It doesn't replace the reduce but it helps to reduce in advance if it's possible. For example the average of 10 numbers if the same than the average of the first 5 (combiner) plus the last 5 (combiner) to produce to outputs and find the average of the last two (reducer)

Hadoop Streaming

Uses Unix standard streams to write MR programs in any language other than Java. A key difference is that the Java API process a map for each record where in streaming you could choose to process more than one line before sending to the output. Another advantage is that is easy to debug as Streaming only uses std IO so it does not need Hadoop to get executed in development.

1. Map input is passed over standard input to process line by line
2. Map output is written as single tab-delimited line
3. Input takes that line passed over standard input
4. Input reads, process and writes to standard output

To run a streaming program you should use the hadoop-*streaming.jar. This is an example: % hadoop jar $HADOOP_INSTALL/contrib/streaming/hadoop*--streaming.jar \ -input input/ncdc/sample.txt \ -output output \ -mapper ch02/src/main/ruby/max_temperature_map.rb \ -combiner ch02/src/main/ruby/max_temperature_reduce.rb \ -reducer ch02/src/main/ruby/max_temperature_reduce.rb

Hadoop Pipes

C++ interface to Hadoop MapReduce using sockets. JNI is not used. Keys and values **ARE** C++ native byte buffers STL strings. They don't work in local mode, only pseudo or fully distributed. To run a pipe:

```
% hadoop pipes \
-D hadoop.pipes.java.recordreader=true \
-D hadoop.pipes.java.recordwriter=true \
-input sample.txt \
-output output \
-program bin/max_temperature
```

Where max_temperature is the compiled C++ file. The record reader and writer specifies to use the Java ones if no other are specified in the C++ source code.

Hadoop Distributed File System HDFS

Good for

- **Very large files** Over hundreds of MB
- **Streaming data access** Write-once, read-many. Time to read whole dataset is more important than time to read the first record
- **Commodity hardware** Doesn't require expensive hardware

Not good for

- **Low-latency data access**
- **Lots of small files**
- **Multiple writers, arbitrary file modifications** No support for multiple writers on the same file

HDFS Concepts

Blocks

Minimum amount of data a disk can read. 64 MB for HDFS. If a file is smaller than a single block, it does not occupy a full block. They are larger to minimize the cost of seeks because time to transfer data from disk is larger than seeking for a file.

Blocks allows to store huge files (bigger than a hard disk) splitting it's block over the cluster, so it also does not need to be stored in the same disk. Blocks also simplifies the storage subsystem and fits well with replication for providing fault tolerance.

```
# To read blocks instead of files in HDFS
hadoop fsck / -files -blocks
```

Namenodes and Datanodes

Namenode maintains filesystem tree and metadata for files in two files: **namespace image** and **edit log**, knows the datanodes a on which all blocks of a file are located, however, it does no store block locations persistently, since this info is reconstructed from datanodes. Without the Namenode it's not possible to reconstruct the filesystem that will be lost so the Namenode can back up the files to multiple filesystems (local and NFS) in synchronous and atomic way. It can also use a **SecondaryNamenode** (on a separate machine because it needs lots of CPU and memory) to merge the namespace with the edit log to prevent it becomes too large. However the state of SecondaryNamenode is not perfectly sincronized in time with Namenode. Still the Namenode is the **single point of failure (SPOF)**.

Datanodes stores and retrieve blocks and report back to Namenode with lists of blocks that they are storing.

HDFS Federation

Allows more Namenodes to manage portions of the filesystem, *namespace volume*, and all its blocks, *block pool*. Clients use client-side mount tables to map file paths to namenodes. This is managed with **ViewFileSystem** and *viewfs://* URI's.

HDFS High Availability

To recover from a failure, new primary namenode is started and configure datanodes and clients to use it. It's not able to ser until:

1. It load the namespace into memory
2. It has replayed it's edit log
3. It has received enought block reports from the datanodes to leave safe mode

HDFS HA is a pair of namenodes in active-standby configuration but it needs few architectural changes:

- Namenodes must **use highly available** shared storage to share the edit log (for example using a BookKeeper based on ZooKeeper)
- Datanodes must **send block reports to both namenodes** since block mappings are stored in memory and not disk
- Clients must be **configured to handle namenode failover**

Failover and fencing

Transition to one namenode to the other is handled by the **failover controller** with Zookeeper. Each namenode runs a process to monitor its namenode for failures. Also, failover can be initiated manually (graceful failover)

In case of failover it is impossible to know that a namenode has stopped running because simply slow network can trigger a failover transition. The HA prevents that an old namenode corrupt the filesystem (**fencing**). The system kill the namenode process, revokes its access to shared storage and disabling its network port. Alternatively it can be phisically power down the host (STONITH = shoot the other node in the head)

The command line interface

Two key properties:

1. **fs.default.name** `hdfs://localhost` used to set a default filesystem for Hadoop. Default port 8020
2. **dfs.replication** `3` Three replicas per HDFS block

Basic filesystem operations

- **hadoop fs -copyFromLocal [input] [output]** copies from local storage to HDFS
- **hadoop fs -copyToLocal [input] [output]** copies from HDFS to local storage
- **hadoop fs -ls [path]** list files in HDFS
- **hadoop fs -mkdir [folder]** creates a new folder in the HDFS

Permissions

Client's identity is the username and group of the process it is running in. The `dfs.permissions` property is enabled by default that checks if the client's username matches the owner and group

The Hadoop Filesystems

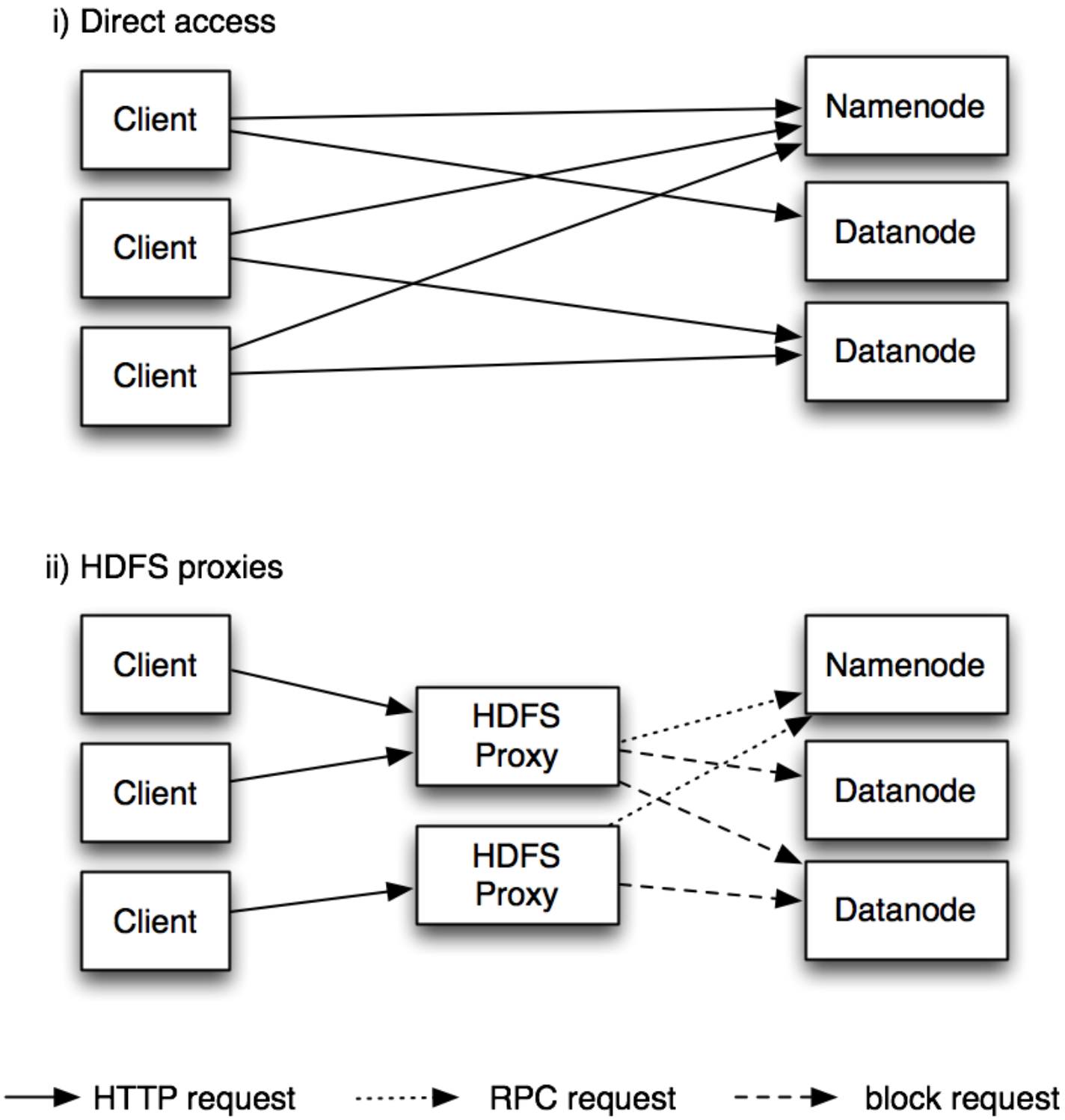
The Java abstract org.apache.hadoop.fs.* has several implementations:

Filesystem	Uri scheme	Java Implementation	Description
Local	file	fs.LocalFileSystem	Locally connected disk with

			checksums
HDFS	hdfs	hdfs.DistributedFileSystem	HDFS
HFTP	hftp	hdfs.HftpFileSystem	Read only access to HDFS over HTTP
HSFTP	hsftp	hdfs.HsftpFileSystem	Read only access to HDFS over HTTPS
WebHDFS	webhdfs	hdfs.web.WedHdfsFileSystem	Secure read write access to HDFS
HAR	har	fs.HarFileSystem	A fs layered over another fs
KFS	kfs	fs.kfs.KosmosFileSystem	CloudStore C++ fs
FTP	ftp	fs.ftp.FTPFileSystem	Fs backed by a FTP server
S3	s3n	fs.s3native.NativeS3FileSystem	Fs of Amazon S3
S3 (block)	s3	fs.s3.S3FileSystem	S3 storing in blocks
Dist. Raid	hdfs	hdfs.DistributedRaidFileSystem	Raid version of HDFS
View	viewfs	viewfs.ViewFileSystem	Client-side mount table for HDFS

Interfaces

HTTP



The first way of accesing, direct, is served directly by namenodes webserver (defaults to 50070) while file data is streamed from datanodes by their web servers (running on port 50075). The WebHDFS must be enabled by setting `dfs.webhdfs.enabled` to true.

The second, throught proxy, is common to transfer data between clusters located in differents data centers. The HttpFs proxy has read-write capabilities and exposes the same interfaces than WebHDFS.

C

The C API mimics the Java lag with some delay in features.

Fuse

Filesystem in Userspace allows filesystems that are implemented in user space to be integrated as a Unix filesystem. Hadoop's Fuse-DFS allows a HDFS fs to be mounted as a standar fs to be used with the typical command line tools.

The Java interface

Reading and writing data with the Filesystem API

With java.net.URL you should also use **setURLStreamHandlerFactory** with an instance of **FsUrlStreamHandlerFactory** (can only be called once per JVM).

Using a URLStreamHanlderFactory you can use the FileSystem API to open a stream of an input HDFS file that is represented as Hadoop Path object (not a java.io.File). It also returns an instance of FSDataInputStream instead of the one of java.io. For writing is similar but using FSDataOutputStream and **create()**.

There are also methods for creating and deleting directories and to query the fs to navigate. This is achieved with the FileStatus. You can list files normally or using regexp. Finally you can also delete files with the Filesystem API

Data flow

Anatomy of a file read

Anatomy of a file write

Replica placement

It places first replica on the same node as the client, second on a different random rack from the first (*off-rack*), the third is on the same rack as the second but on a different node

Coherency model

Describes data visibility or reads and writes because any content written to files are not guaranteed to be visible even if the stream is flushed until at least one block of data is written.

It is possible to force all buffers to be synchronized to the datanodes via **FSDataOutputStream.sync()**

Without the sync a full block is lost in case of failure. If this is not acceptable a sync() must be placed after writing a certain number of records

Parallel Copying with distcp

To copy large amounts of data to and from Hadoop. Sintax:

```
hadoop distcp [hdfs://input/data] [hdfs://new-location]
```

distcp is a MapReduce job without reducers, the numbers of map is total size divided by 256 (in MB). 20 map task are default for each TaskTracker. It can not copy between different cluster versions.

Hadoop Archives

Many 1mb small size files will use a 64 mb block each (even when it will still use the 1mb). To handle this a Hadoop HAR file can be created collecting many small files that is made of:

1. part-* -> Contains the content of the original files concatenated together
2. _index -> Use to search for a file inside the part- files
3. _masterindex

However, create a HAR **creates a copy of the original files** and to add or remove it's needed to re-create the HAR file.

Hadoop I/O

Data integrity in HDFS

HDFS transparently checksums all data for every io.bytes.per.checksum bytes of data (512 bytes). When writing or reading to a pipeline of datanodes, the last verifies checksum.

HDFS can "heal" corrupted blocks with the replicas. It is also possible to disable verification passing `false` to the `FileSystem.setVerifyChecksum()` or throught the command line using `-ignoreCrc`

LocalFileSystem

Hadoop creates a hidden *.crc file using ChecksumFileSystem everytime it writes a file with a chunk size set in io.bytes.per.checksum

Compression

When compressed, inputs will be decompressed automatically by MapReduce. To compress the output, set the `mapred.output.compress` to true and the `mapred.output.compression.codec` to the classname of the compression format you want to use

Compression format	Tool	Algorithm	Extension	Splittable
DEFLATE	N/A	DEFLATE	.deflate	No
gzip	gzip	DEFLATE	.gz	No
bzip2	bzip2	bzip2	.bz2	Yes
LZO	lzop	LZO	.lzo	Yes/No
Snappy	N/A	Snappy	.snappy	No

Compression and Input Splits

Splittable compression formats allow to search inside 16-blocks file when having only one of them locally. Depending on the application, one format is better suited than others. When storage costs are critical, you should use the highest compression even if it doesn't split and the same mapper has to process the 16 blocks. It can even compress the map output

Property name	Type	Default value	De
mapred.output.compress	boolean	false	Cor outp
mapred.output.compression.codec	Class name	org.apache.hadoop.io.compress.DefaultCodec	The com cod for
mapred.output.compression.type	String	RECORD	The com to u Sec

Serialization

Is turning structured objects into a byte stream for transmission over a network or writing to persistent storage. Deserialization is turning a byte stream back into a series of structured objects. Is used for interprocess communication (RPC) and for persistent storage.

RPC must be:

1. Compact
2. Fast
3. Extensible (evolve straightforward)
4. Interoperable (support clients written in diff languages)

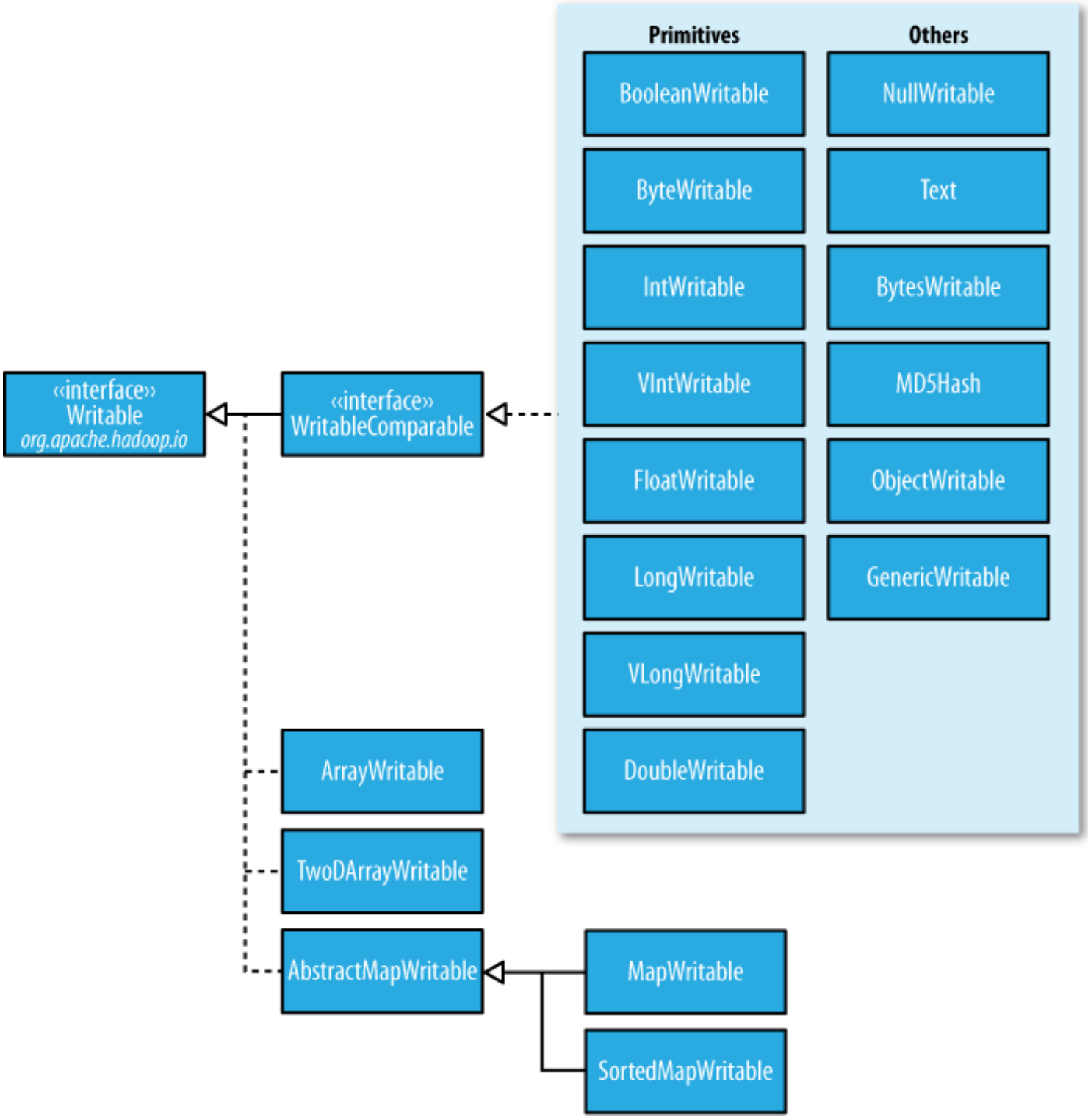
The Writable Interface

Writable and WritableComparable

2 methods for Writable interface: **DataOutput** and **DataInput**. WritableComparable is a subinterface of

Writable and java.lang.Comparable. Comparisons are crucial for sorting.

Writable Classes



When encoding integers you can use the fixed length (IntWritable) or variable (VIntWritable). Most numeric variables tend to have non uniform distributions so on average they will save space.

Text

Equivalent to Java String, maximum value is 2gb as it stores the number of bytes in an int.

BytesWritable

Wrapper for an array of binary data serialized as an 4-byte integer that specifies the bytes to follow and the bytes themselves. For example, length=2 (4 byte integer 00000002) with values 3 and 5 (03 and 05) == 000000020305

NullWritable

Zero length, singleton

ObjectWritable and GenericWritable

Wrapper for String, enum, Writable and null or arrays of any of them. **ObjectWritable** is useful when a field can be more than one type, for example in a SequenceFile with multiple types. If the number of types is known in advance, you can write an array of the classes and use **GenericWritable** instead

Writable Collections

- ArrayWritable and TwoDArrayWritable, for arrays and 2d Writables. They must all be the same instances of the same class.
- ArrayPrimitiveWritable wrapper for Java primitives arrays.
- MapWritable (java.utilMap) and SortedMapWritable (java.util.SortedMap) EnumSetWritable.

Implementing a Custom Writable

1. It's needed to extend **WritableComparable** to have the **compareTo()**
2. Have an empty constructor
3. Populates the fields calling **readFields()
4. Have a **write()** method
5. Override **hashCode(), equals()** and **toString()**

Serialization Frameworks

A Serialization defines a mapping from types to Serializer instances (for turning an object into a byte stream)

Avro

Resume in here: http://blog.cloudera.com/blog/2009/11/avro-a-new-format-for-data-interchange/

Language-neutral serialization system to be processed by many languages (C, C++, Python...). Uses *schemas* in JSON but code generation is optional (you can read data that conforms to a given schema even

if your code has not seen that schema before) and provides API's for des/serialization

It has common data types (null, boolean, int, bytes, string...) and some complex types (array, map, enum...). Typical Avro schema with types:

```
{
    "type":"record",
    "name":"StringPair",
    "doc": "A pair of strings.",
    "fields":{
        {"name":"left", "type":"string"},
        {"name":"right", "type":"string"}
    }
}
```

Then to load, use and serialize a record in Java (the use will be similar in Python, for example):

```
//Loads
Schema.Parser parser = new Schema.Parser();
Schema schema = parser.parse(getClass().getResourceAsStream("StringPair.avsc"));

//Creates Avro instance
GenericRecord datum = new GenericData.Record(schema);
datum.put("left", "L");
datum.put("right", "R");

//Serialize to an output stream
ByteArrayOutputStream out = new ByteArrayOutputStream();
DatumWriter<GenericRecord> writer = new GenericDatumWriter<GenericRecord>(schema);
Encoder encoder = EncoderFactory.get().binaryEncoder(out, null);
writer.write(datum, encoder);
encoder.flush();
out.close();
```

TODO?

File-Based Data Structures

SequenceFile

- **Writing**: Good to persist binary data structure for key-value pairs. To write a SequenceFile use a **createWriter()** method which returns a SequenceFile.Writer where we use the **append()** method (optional arguments include compression and codec).

- **Reading**: With a **SequenceFile.Reader** and iterating with **next()** if they are Writable types or also **getCurrentValue(Object val)** if using non-Writable.
- **Seeking / finding**: Using **seek()** to retrieve a given position in a sequence file.

Command line actions

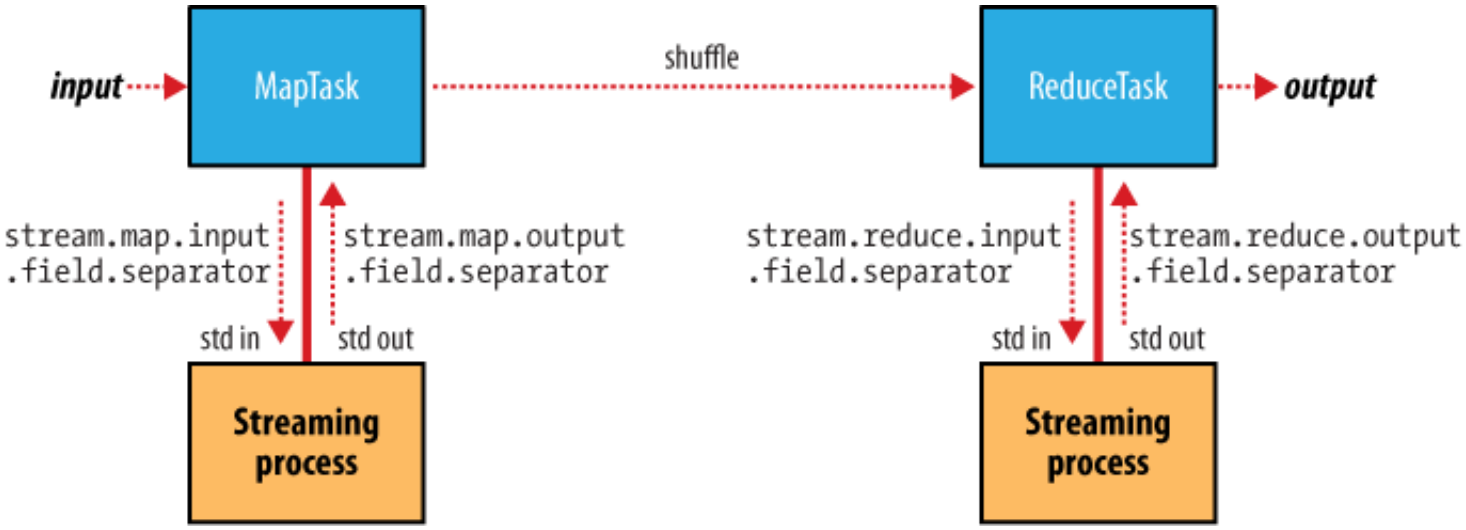
If a SequenceFile has a meaningful text representation it can be read as follows:

```
hadoop fs -text numbers.seq | head
```

And can be sorted using MapReduce job

The SequenceFile format

Consists of a header followed by one or more records, it also contains other fields including the *sync* marker that allow a reader to synchronize to a record boundary.



MapFile

Is a sorted SequenceFile with an index por permit lookups by key. Is written with an instance of **MapFile.Writer** and calling **append()** to add entries in order.

Developing a MapReduce Application

Configuration API

A Configuration class is used to access the configuration XML and can be combined (if a var is repeteated, last is used). Variables can also be expanded using system properties.

Configuring the Development Environment

All JAR's from top level Hadoop directory must be added to the IDE. Also, you can have local and cluster file configurations.

GenericOptionsParser, Tool and ToolRunner

- **GenericOptionsParser** interprets Hadoop command-line options and sets them on a Configuration object
- **Tool** is an interface to use the above class

The `-D` option takes priority over Configuration files

Writing Unit Tests

Mapper Unit Test

Because Mapper and Reducers writes to Context files (instead of returning the result) a mock for the Context object is needed. We use Mockito as follows:

```
@Test
    public void processesValidRecord() throws IOException, InterruptedException {
        MaxTemperatureMapper mapper = new MaxTemperatureMapper();
        Text value = new Text(
```

```
            "00430119909999919500515180004+68750+023550FM-12+038299999V0203201N00261220001CN9999999
            );

        MaxTemperatureMapper.Context context = mock(MaxTemperatureMapper.Context.class);

        mapper.map(null, value, context);

        verify(context).write(new Text("1950"), new IntWritable(-11));
    }
}
```

We create the context object passing to the static `mock` method the class. Then we use it normally. Reducer unit test is similar.

Running locally and in a cluster on Test Data

- **Locally** Using the Tool interface you could write a driver to configure the local job.
- **Cluster** No code changes are needed, just to pack the Jar.

The MapReduce Web UI

Jobtracker Page

ip-10-250-110-47 Hadoop Map/Reduce Administration

Quick Links

State: RUNNING
Started: Sat Apr 11 08:11:53 EDT 2009
Version: 0.20.0, r763504
Compiled: Thu Apr 9 05:18:40 UTC 2009 by ndaley
Identifier: 200904110811

Cluster Summary (Heap Size is 53.75 MB/888.94 MB)

Maps	Reduces	Total Submissions	Nodes	Map Task Capacity	Reduce Task Capacity	Avg. Tasks/Node	Blacklisted Nodes
53	30	2	11	88	88	16.00	0

Scheduling Information

Queue Name	Scheduling Information
default	N/A

Filter (Jobid, Priority, User, Name)
Example: 'user:smith 3200' will filter by 'smith' only in the user field and '3200' in all fields

Running Jobs

Jobid	Priority	User	Name	Map % Complete	Map Total	Maps Completed	Reduce % Complete	Reduce Total	Reduces Completed	Job Scheduling Information
job_200904110811_0002	NORMAL	root	Max temperature	47.52%	101	48	15.25%	30	0	NA

Completed Jobs

Jobid	Priority	User	Name	Map % Complete	Map Total	Maps Completed	Reduce % Complete	Reduce Total	Reduces Completed	Job Scheduling Information
job_200904110811_0001	NORMAL	gonzo	word count	100.00%	14	14	100.00%	30	30	NA

Failed Jobs

none

Local Logs

Log directory, Job Tracker History

Hadoop, 2009.

1. Hadoop installation: version, compilation, jobtracker state...
2. Summary of the cluster: capacity, utilization, mr running, jobs, tasktrackers, slots, blacklisted Tasktrackers
3. Job Scheduler: Running and failed jobs with id's, owner, name...
4. Link to Jobtracker Logs: historic

Job page

Hadoop job_200904110811_0002 on ip-10-250-110-47

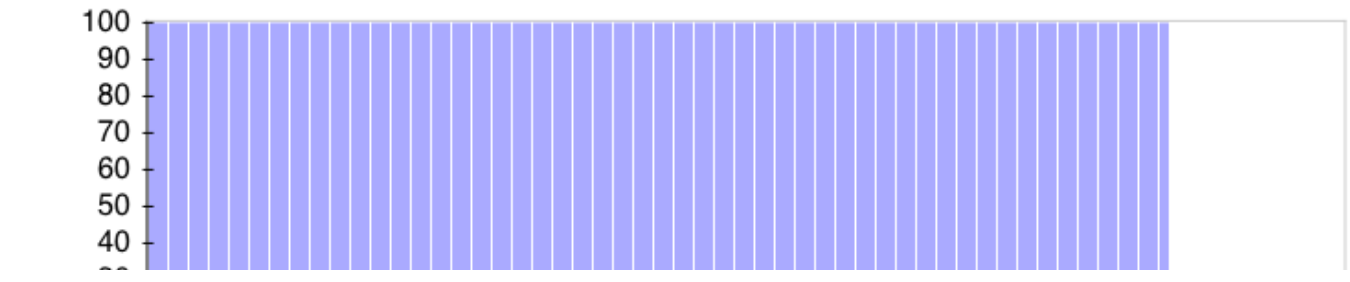
User: root

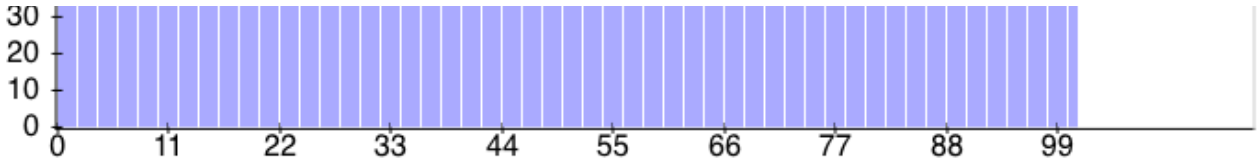
Job Name: Max temperature
Job File: hdfs://ip-10-250-110-47.ec2.internal/mnt/hadoop/mapred/system/job_200904110811_0002/job.xml
Job Setup: Successful
Status: Running
Started at: Sat Apr 11 08:15:53 EDT 2009
Running for: 5mins, 38sec
Job Cleanup: Pending

Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	100.00%	101	0	0	101	0	0 / 26
reduce	70.74%	30	0	13	17	0	0 / 0

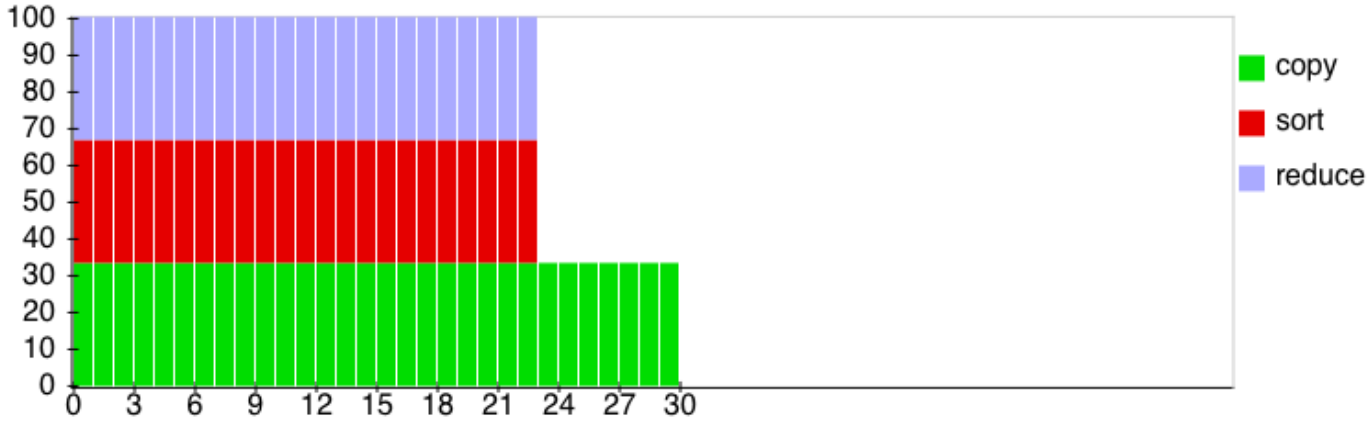
	Counter	Map	Reduce	Total
Job Counters	Launched reduce tasks	0	0	32
	Rack-local map tasks	0	0	82
	Launched map tasks	0	0	127
	Data-local map tasks	0	0	45
FileSystemCounters	FILE_BYTES_READ	12,665,901	564	12,666,465
	HDFS_BYTES_READ	33,485,841,275	0	33,485,841,275
	FILE_BYTES_WRITTEN	988,084	564	988,648
	HDFS_BYTES_WRITTEN	0	360	360
Map-Reduce Framework	Reduce input groups	0	40	40
	Combine output records	4,489	0	4,489
	Map input records	1,209,901,509	0	1,209,901,509
	Reduce shuffle bytes	0	18,397	18,397
	Reduce output records	0	40	40
	Spilled Records	9,378	42	9,420
	Map output bytes	10,282,306,995	0	10,282,306,995
	Map input bytes	274,600,205,558	0	274,600,205,558
	Map output records	1,142,478,555	0	1,142,478,555
	Combine input records	1,142,482,941	0	1,142,482,941
	Reduce input records	0	42	42

Map Completion Graph - close





Reduce Completion Graph - close



Go back to JobTracker

Hadoop, 2009.

- Job progress
- Owner
- Name
- Running time
- Completion graphs

Retrieving the results

Each map will write a single file. The `-getmerge` option of `hadoop fs` gets all files in a folder and merges the into a single local file.

Debugging a Job: The tasks page

It is often useful to use a counter in the MR. Write a MR to read logs or write info to map output that can be check on the tasks page in the "status" column. The Action column allows to kill a task if `webinterface.private.actions` is set to true.

Hadoop Logs

Logs	Primary audience	Description
System daemon logs	Administrators	Each Hadoop daemon produces a logfile (using log4j) and another file that combines standard out and error. Written in the directory defined by the HADOOP_LOG_DIR environment variable
HDFS audit logs	Administrators	A log of all HDFS requests, turned off by default. Written to the namenode's log, although this is configurable
MapReduce job history logs	Users	A log of the events (such as task completion) that occur in the course of running a job.Saved centrally on the jobtracker, and in the job's output directory in a _logs/history subdirectory
MapReduce task logs	Users	Each tasktracker child process produces a logfile using log4j (called syslog), a file for data sent to standard out (stdout), and a file for standard error (stderr). Written in the userlogs subdirectory of the directory defined by the HADOOP_LOG_DIR environment variable

Logfiles can be found on the local fs of each TaskTracker and if JVM reuse is enabled, each log accumulates the entire JVM run.

Anything written to standard output or error is directed to the relevant logfile.

Remote debugging

No direct ways. Options:

- **Reproduce the failure locally** Download the fail that makes the task to fail.
- **Use JVM debugging options** -XX:-HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/path/to/dumps to log Java out of memory errors.
- **Use task profiling** Mechanism to profile a subset of the task

Sometimes is useful to keep intermediate files for a failed task setting `keep.failed.task.files` to *true* that will store the files in the `mapred.local.dir` of the node.

Tuning a Job to improve performance

| Area | Best Practice | | ---------- + ---------- | | Number of mappers | How long are your mappers running for? If they are only running for a few seconds on average, then you should

see if there's a way to have fewer mappers and make them all run longer, a minute or so, as a rule of thumb. The extent to which this is possible depends on the input format you are using | | Number of Reducers | For maximum performance, the numberof reducers should be slightly less than the number of reduce slots in the cluster. This allows the reducers to finish in one wave and fully utilizes the cluster during the reduce phase | | Combiners | Can your job take advantage of a combiner to reduce the amount of data in passing through the shuffle? | | Intermediate Compression | Job execution time can almost always benefit from enabling map output compression | | Custom serialization | If you are using your own custom Writable objects, or custom comparators, then make sure you have implemented RawComparator | | Shuffle Tweaks | The MapReduce shuffle exposes around a dozen tuning parameters for memory management, which may help you eke out the last bit of performance |

Profiling tasks

HPROF is enabled in JobConf:

```
Configuration conf = getConf();
conf.setBoolean("mapred.task.profile", true);   // Enable profiling
conf.set("mapred.task.profile.params", "-agentlib:hprof=cpu=samples," +
    "heap=sites,depth=6,force=n,thread=y,verbose=n,file=%s");   //Params
conf.set("mapred.task.profile.maps", "0-2"); // Tasks to profile
conf.set("mapred.task.profile.reduces", ""); // no reduces
Job job = new Job(conf, "Max temperature");
```

MapReduce Workflows, job control

For a linear chain, use JobClient like `JobClient.runJob(conf1)`, `JobClient.runJob(conf2)`, etc. If a job fails will throw an IOException. You could also use JobControl from the client machine to represents a graph of jobs.

Apache Oozie

1. **Workflow engine**: Stores and runs workflows composed of Hadoop jobs.
2. **Coordinator engine**: Run workflows jobs based on predefined schedules and data availability

Oozie runs as a service in the cluster receiving workflows (DAG) of **action nodes** (moving files in HDFS, running MR, Pig...) and **control flow nodes** (flow between action nodes).

- **Workflow**
    - One **start** and **one** end node.

- `Kill` nodes finished a workflow as failed and reports the message specified
- A **map-reduce** action
  - **job-tracker** Specifies jobtracker to submit
  - **name-node** URI for input/output data
  - A **configuration** element to specofy key/value pairs
- A **prepare** action is executed before the map-reduce

Running an Oozie workflow job

```
$ export OOZIE_URL="http://localhost:11000/oozie"
$ oozie job -config [*.properties] -run
```

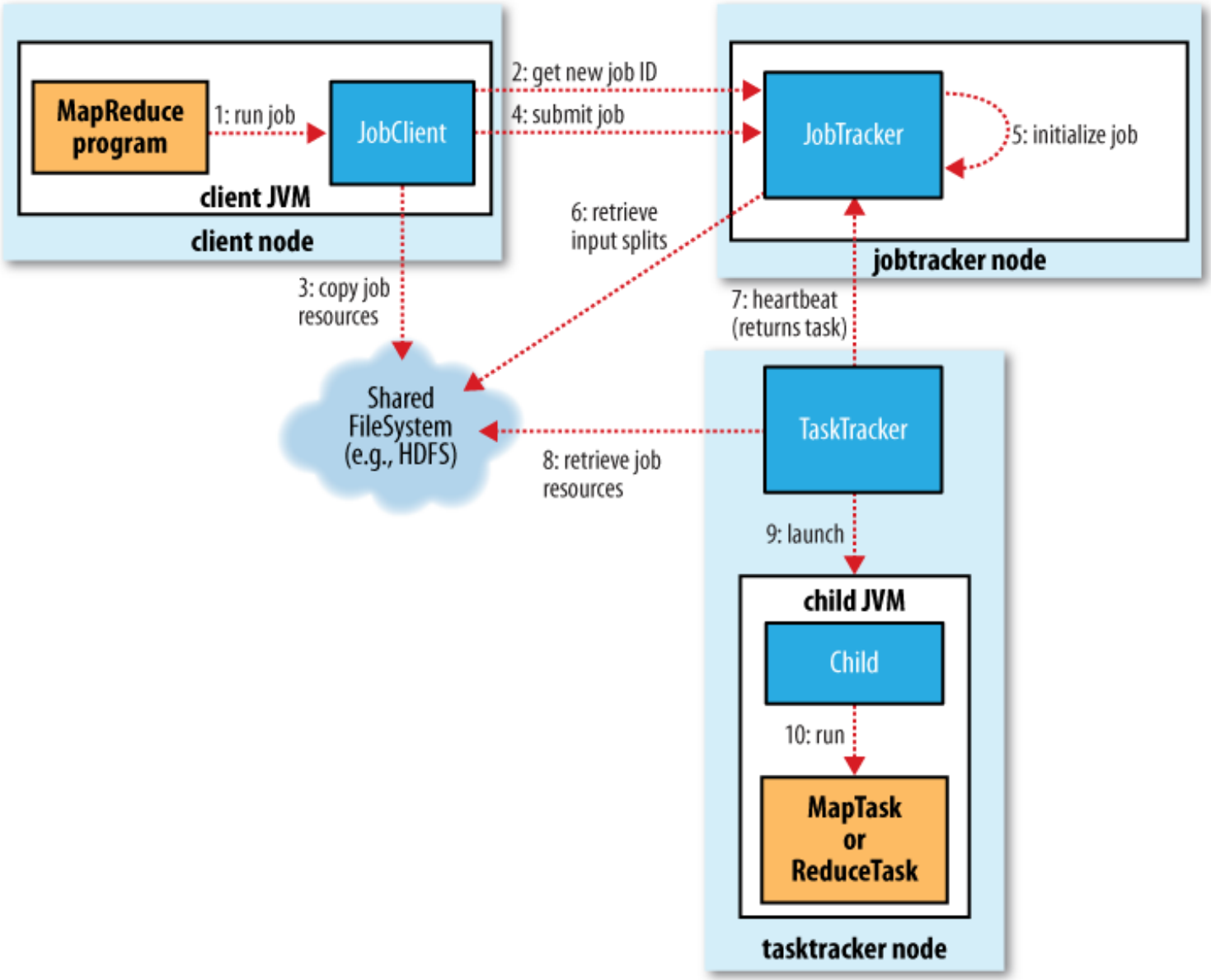
`-config` defines local Java properties in the workflow XML as well as `oozie.wf.application.path` which tells Oozie workflow app. `-info` gives info about the workflow job. A properties file:

```
nameNode=hdfs://localhost:8020
jobTracker=localhost:8021
oozie.wf.application.path=${nameNode}/user/${user.name}/max-temp-workflow
```

How MapReduce Works

Prior to 0.23, setting `mapred.job.tracker` to **local** the local job runner is used. YARN uses `mapreduce.framework.name` which takes the values **local**, **classic** and **yarn**

Classic MapReduce (MapReduce 1)



1. Client submits job
2. **JobTracker** coordinates job run

3. **TaskTracker** run the tasks that the job has been split into
4. The **HDFS** is used for sharing job files.

submit() method on **Job** creates an internal **JobSummiter** and calls **submitJobInternal()** on it:

1. Asks jobtracker for new job ID calling **getNewJobId()** on **JobTracker**
2. Checks output dir
3. Computes input splits
4. Copies the needed resources (JAR, config file...). The JAR is copied with high replication factor set by `mapred.submit.replication` property (defaults to 10)
5. Tells the **JobTracker** the job is ready calling **submitJob()**

Job Initialization

JobTracker enqueue the job and creates a list of tasks retrieving the input splits to create one map for each split. The number of reduce tasks to create is in `mapred.reduce.tasks` or set by the **setNumReduceTasks()**. Also a *job setup tasks* and a *job cleanup task* are run by the **TaskTrackers** to setup the job and cleanup after reduce tasks are completed. **OutputCommiter** the code to run

Task Assignment

TaskTrackers tells JobTracker they are alive and if they are ready throught *heartbeat* calls and they have a fixed number of slots for map and reduce tasks. Map slots has priority over reduce ones.

The **JobTracker** picks a map task whose input split is as close as possible to the tasktracker If possible it will be *data-local*. Alternatively, *rack-local* (same rack). To choose a reduce it simply takes the next in the queue.

Task Execution

The JAR is copied to the tasktracker's fs, creates a local working dir and a instance of TaskRunner

TaskRunner launches a new JVM (it can be re-used by tasks but shouldn't be shared with the TaskRunner). Each task can perform setup and cleanup actions determined by the OutputCommiter

Progress and Status Updates

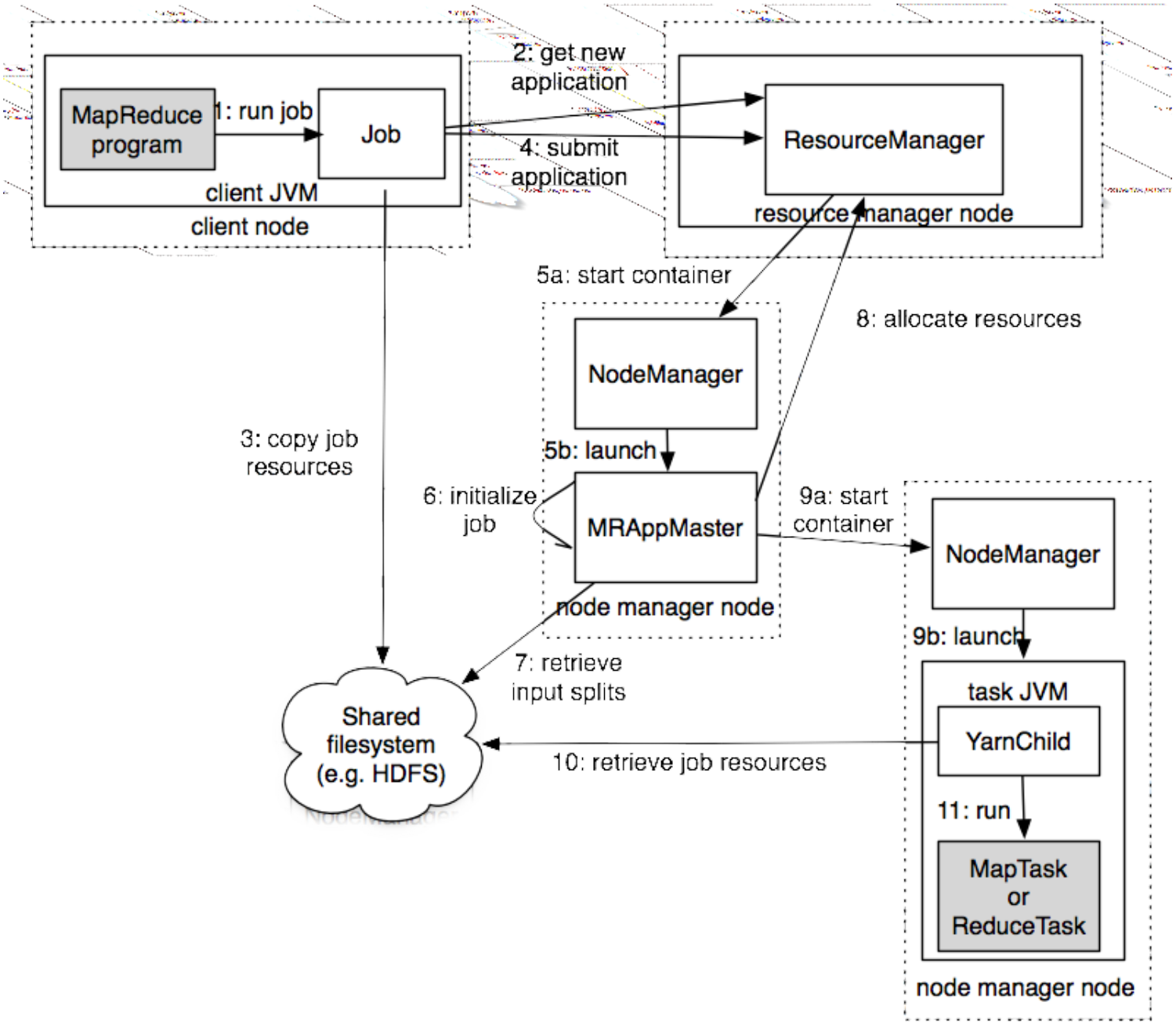
A job and its tasks have a status. The progress in the map is the proportion of the input that has been processed. For reducer is an estimation of the portion of the reduce. It can also be counters.

Clients can use **Job.getStatus()** to obtain a **JobStatus** instance

Job completion

The jobtracker changes the status of the job to "successful", the **Job** polls for status and prints a message and returns from **waitForCompletion()**. It also sends a HTTP job notification if `job.end.notification.url` is set

YARN (MapReduce 2)



Splits JobTracker into 2: a **resource manager** to manage the use of resources across the cluster and an

application master to manage the lifecycle of the apps running the cluster. The idea is that an app negotiates with the *resource manager* for cluster resources. Each instance of an app has an **application master** that runs for the duration of the app.

MR on YARN has:

- The client which submits the MR job
- The YARN resource manager coordinates allocation of compute resources
- The YARN node managers: launch an monitor nodes
- The MR application master which coordinates tasks running the MR job. Application master and MR tasks run in containers that are scheduled by the resource manager and managed by the node manager.
- The distributed filesystem

Job submission and initialization

Job is submitted calling **submitApplication()** and generates a job id, however it is now called application ID

The **application master (MRAppMaster)** initializes the job, retrieves input splits, create a map task for each split and the number of reduces set in `mapreduce.job.reduces`.

AppMaster then run the job in the same JVM if it's small (less than 10 mappers, only one reducer and input size is less than the size of one HDFS block) and it doesn't allocate too many containers and running tasks. This is called **uber task**.

Uber Task property
mapreduce.job.ubertask.maxmaps
mapreduce.job.ubertask.maxreduces
mapreduce.job.ubertask.maxbytes
mapreduce.job.ubertask.enable

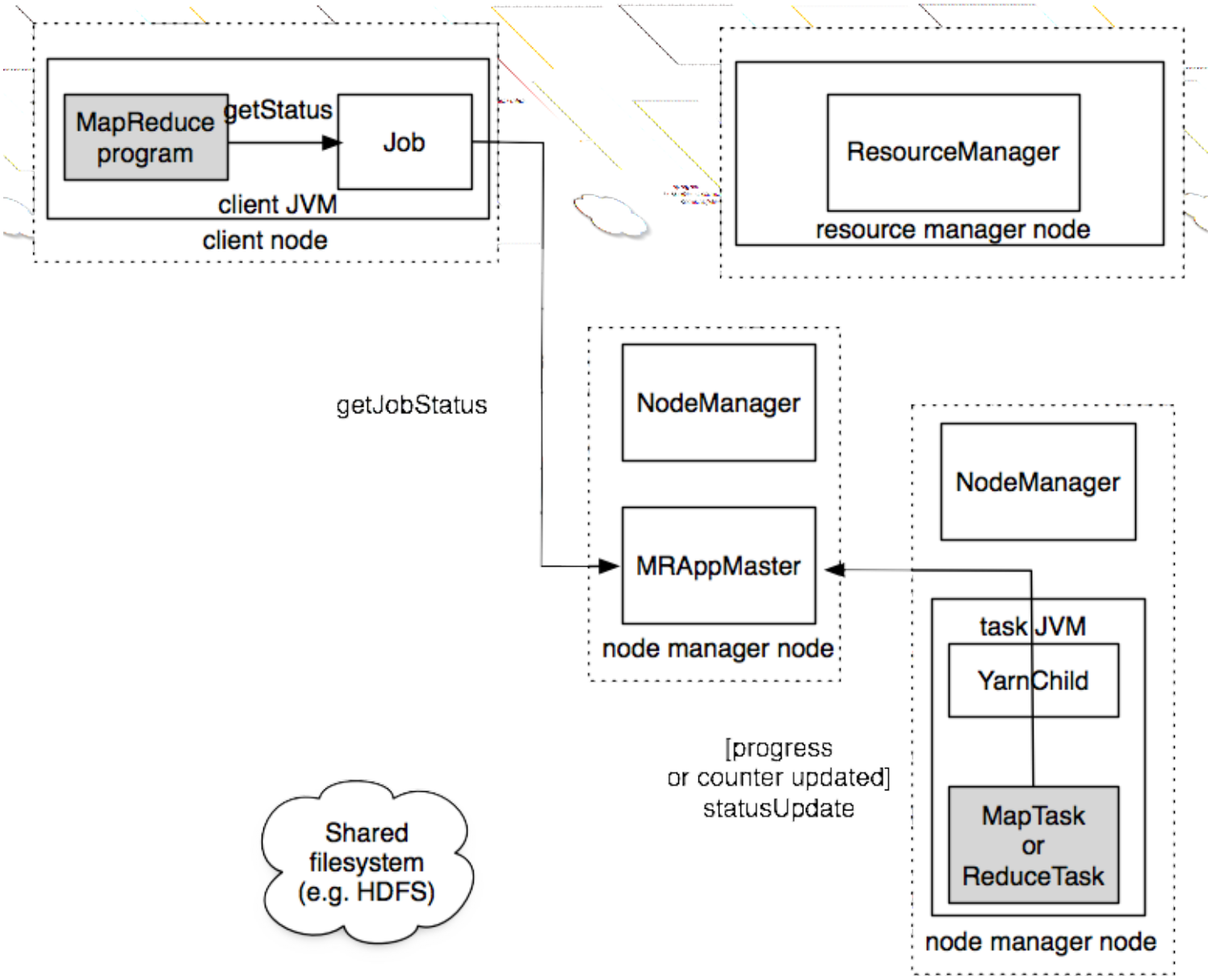
Task Assignment

AppMaster requests containers from **Resource manager** and memory, configurable throught `mapreduce.map.memory.mb` and `mapreduce.reduce.memory.mb`

The memory must be multiple of minimum allocation and it will request just the needed amount: 1024 mb

`yarn.scheduler.capacity.minimum-allocation-mb` and the default maximum is 10240 `yarn.scheduler.capacity.maximum-allocation-mb`. So it'll request between 1gb and 10gb

Task execution



YarnChild localizes resources and runs in a dedicated JVM (but it can't be reused)

Progress and Status Updates

Reports every 3 seconds to AppMaster

Job Completion

Every 5 seconds (configured in `mapreduce.client.completion.pollinterval`) client checks for completion usin **waitForCompletion()**

Failures

Classic MapReduce

Task Failure

JVM reports error to TaskTracker and exists. For Streaming a non-zero exit is a failure (configured in `stream.non.zero.is.failure`)

10 minutes (`mapred.task.timeout`) is the timeout for hanging. If it's passed the tastracker marks the task as failed. Timeout to zero disables timeout.

When **JobTracker** is notified of a task failure, it will reschedule the task in a different TaskTracker. If a task fails 4 times (`mapred.map.max.attempts` and `mapred.reduce.max.attempts`) **the whole job fails**. It can also be configured as a maximum allowed percentage (`mapred.max.map.failurs.percent` and `mapred.max.reduce.failures.percent`)

Tasks can also be killed throught command line and the Web UI

TaskTracker Failure

JobTracker notice a timeout failure (`mapred.task.tracker.expiry.interval`) on a tasktracker and removes it from the pool.

Fault occurs when `mapred.max.tracker.failures` pass 4 task failures on the same job. 4 faults (`mapred.max.tracker.blacklists`) **blacklists** the tasktracker. Faults expire one per day. Restarting a tasktracker makes it abandon the blacklist

JobTracker Failure

Single point of failure = The job fails. Jobs will need to be resubmitted

Failures in YARN

Task Failure

Similar to classic.

Application Master failure

Marked as failed if they fail once (`yarn.resourcemanager.am.max-retries`). They can be recover if `yarn.app.mapreduce.am.job.recovery.enable` is set to true

Node Manager Failure

Stops sending heartbeat and is removed from resource manager (timeout set in `yarn.resourcemanager.nm.liveness-monitor.expiry-interval-ms).

Node managers are blacklisted if the numbers of failures for the application is high (`mapreduce.job.maxtaskfailures.per.tracker`)

Resource manager failure

Uses a checkpointing mechanism to save it's state (node managers and applications) and it is brought up (by an administrator).

The storage is configured via `yarn.resourcemanager.store.class` , by default it keeps in memory but ZooKeeper can be used

Job Scheduling

Follows a FIFO approach, however with `mapred.job.priority` or **setJobPriority()** is used on **JobClient** to set he priority of a Job that the job scheduler knows which to choose next. However a low-priority long-running job will still block the rest (does not support *preemption*)

Fair Scheduler

If a single job is running, it gets the whole cluster, as more jobs are submitted free tasks slots are given to five each user a fair share.

It supports preemption so if a task has not received its fair share for some time, the scheduler will kill tasks in pool to share some resources.

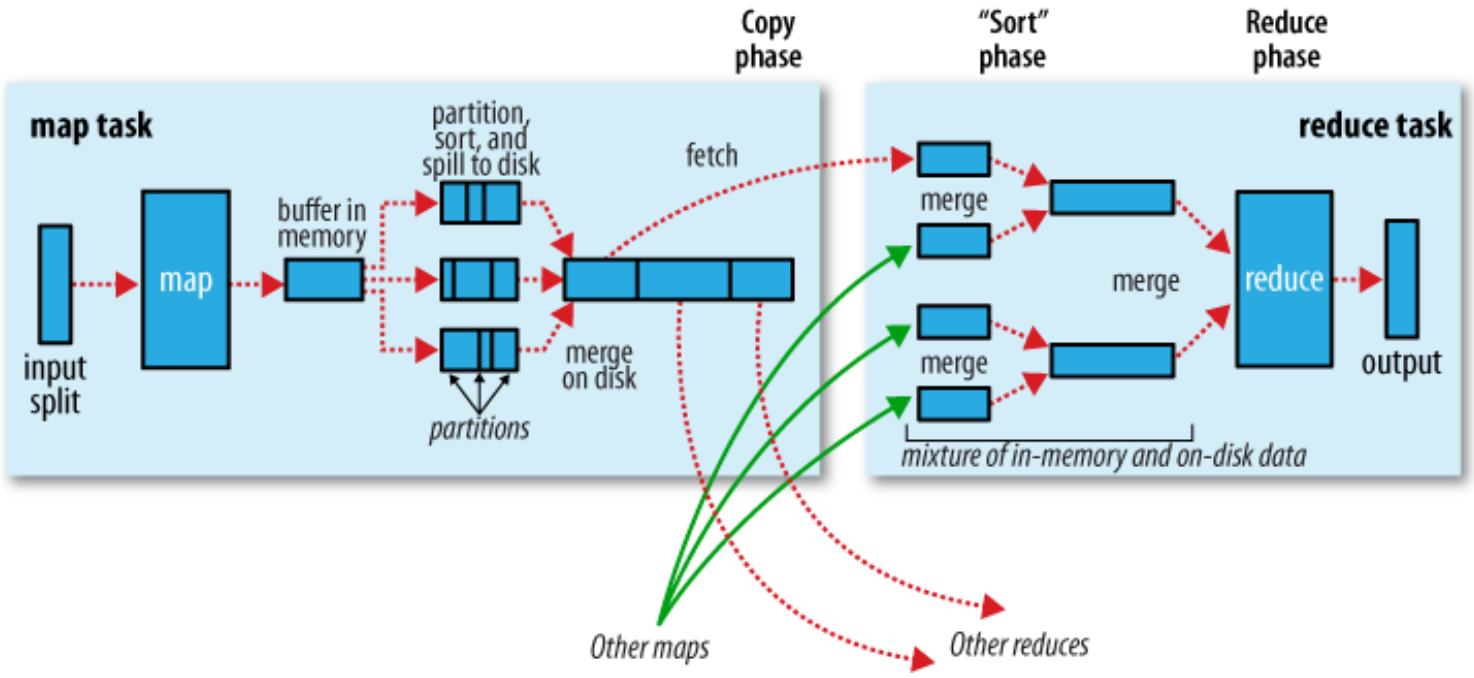
Capacity Scheduler

Allows to separate a MapReduce Cluster with FIFO scheduling

Shuffle and Sort

The input to every reducer is sorted by key, this is known as the *shuffle*

The Map Side



Each map task has a circular memory buffer that it writes the output to. The buffer is 100 MB (`io.sort.mb`). When the content reaches a threshold (`io.sort.spill.percent`, default to 0.80) a background thread will spill the contents to disk. If the buffer is filled the map will block unil the spill is complete and a new spill file is created. Finally all the spills are merged and sorted again. `io.sort.factor` controls the maximum number of streams to merge at once (defaults 10)

Spills are written to the dirs in `mapred.local.dir` but before they are partitioned in as many reducers as they will be sent to. Within each partition a background thread performs an in-memory sort by key and if there is a combiner, it is run on the output.

If there are at least 3 spills files (`min.num.spills.for.combine`) the combiner runs again. It is possible also to compress the output (`mapred.compress.map.output` to **true**).

The output's are sent throught HTTP with a maximum of `tasktracker.http.threads` per tasktracker (not per map)

The Reduce Side

Reduce tasks needs map output from several map tasks so reduce starts copying (with 5 threads as default `mapred.reduce.parallel.copies`) as soon as each map finish (*copy phase*).

Map outputs are copied to the reduce tasks JVM memory if they are small enough (`mapred.job.shuffle.input.buffer.percent`). When the in memory buffer reaches a threshold size (`mapred.job.shuffle.merge.percent`) or reaches a thresold number of map outputs (`mapred.inmem.merge.thresold`), it is merged and spilled to disk. Then they are merged into larger, sorted files.

When all map output have been copied, the reduce move to **sort** phase (properly called the *merge* phase) that merge map outputs maintaining their sort ordering. With 50 map outputs and a *merge factor* of 10 (`io.sort.factor`), each 10 files will output 1 intermediate file.

Lastly, the *reduce phase*, when it's feed of the previous outputs, the one that is written to HDFS.

Configuration Tuning

General rule: give shuffle as much memory as possible. The amount given to JVM is `mapred.child.java.opts`. Avoid multiple spills on map. Try that the intermediate data on reduce reside entirely in memory, by default this does not happen because all memory is reserved for reduce but if reduce has light memory requirements, setting `mapred.inmem.merge.threshold` to **0** and `mapred.job.reduce.input.buffer.percent` to **1.0** may bring a performance boost.

You could also increase the hadoop's buffer size from 4kb to `io.file.buffer.size` property.

Task Execution

MapReduce can obtain information of the environment (like file in process name) and read streaming environment variables.

Speculative Execution

When a job is waiting for a task to finish the job, the cluster launches the same tasks on different nodes to get the result form the fastest one (and discard the rest).

Speculative execution properties

Property	Type	Defaul	Description
mapred.map.tasks.speculative.execution	boolean	true	Activated for map Tasks
mapred.reduce.tasks.speculative.execution	boolean	true	Activated for reduce tasks
yarn.app.mapreduce.am.job.speculator.class	Class	a class	Class implementing the speculative execution policy
yarn.app.mapreduce.am.job.task.estimator.class	Class	a class	Provides estimates for task runtimes

The only reason to turn it off is to improve cluster efficiency, to avoid reduce tasks to fetch the same map outputs over the network and for tasks that are not idempotent.

Output Committers

Used to ensure that jobs and tasks either succeed or fail throught the OutputCommiter class and methods. This are:

- **setOutputCommiter()** on JobConf
- **mapred.output.committer.class**
- **getOutputCommiter()**

Task side-effect files

When writing custom outputs from reduce to HDFS, care needs to be taken that multiple instance don't try to write the same file.

Task JVM Reuse

Jobs with many short-lived tasks can see performance gains with reusing but the task will run sequentially instead of concurrently

Skipping bad records

Skipping record will re-try a task this way:

1. Task fail
2. Task fail
3. **Skipping record** enabled after to fails. Task fail but failed records are stored in the TaskTracker
4. With skipping mode enabled, task succeeds by skipping the records of the 3rd pass.

MaReduce Types and Formats

MapReduce types

- map: (k1, v1) -> list(k2, v2)
- combine: (k2, list(v2)) -> list(k2, v2)
- reduce: (k2, list(v2)) -> list(k3, v3)

Context objects are used for emitting key-value pairs. While it's good to match map output with reduce input, it's not enforced by the Java compiler

The partition operates on the intermediate key and value types (k2 and v2) and returns the partition index.

Input types are set by the input format, the other types are set in the Job. If not set, the intermediates types default to the final output types, so if k2 and k3 are the same there is no need to call **setMapOutputKeyClass()**. Some must be set because some in some aspects the type can be checked at compile time.

The default MapReduce Job

When running a Job without setting a mapper nor reducer it will save on part-r-00* files an integer followed by a tab character and the original data. If we configure the same defaults it will be like the following Job:

```
    @Override
    public int run(String[] args) throws Exception {
        Job job = JobBuilder.parseInputAndOutput(this, getConf(), args);

        job.setInputFormatClass(TextInputFormat.class);

        job.setMapperClass(Mapper.class);

        job.setMapOutputKeyClass(LongWritable.class);
        job.setMapOutputValueClass(Text.class);

        job.setPartitionerClass(HashPartitioner.class);

        job.setNumReduceTasks(1);
```

```
        job.setReducerClass(Reducer.class);

        job.setOutputKeyClass(LongWritable.class);
        job.setOutputValueClass(Text.class);

        job.setOutputFormatClass(TextOutputFormat.class);

        return job.waitForCompletion(true) ? 0 : 1;
    }
```

We set the input format as **TextInputFormat** which produces **LongWritable** (current line in file) and **Text** values. The integer in the final output is actually the line number. The map is the default **Mapper** that writes the same input key and value, by default **LongWritable** as input and **Text** as output.

The partitioner is **HashPartitioner** that hashes the key to determine which partition belongs in. There are as many partitions as reducers and we have only one in the default but if not all records will be evenly allocated across reduce tasks and they will share the same key.

The number of map tasks is equal to the number of splits that the input is turned into. The number of reducers will be equal to the number of nodes multiplied by the slots per node `mapred.tasktracker.reduce.tasks.maximum`. It it's good to have slightly fewer reducers than total slots.

The output key is **LongWritable** and the output value is **Text**. Records are sorted before the reducer.

The default Streaming Job

```
    $ hadoop jar $HADOOP_INSTALL/contrib/streaming/hadoop*-streaming.jar \
    -input input/sample.txt \
    -output output \
    -mapper /bin/cat
```

There is not default identity mapper so it must explicitly be set. Hadoop Streaming output keys and values are always Text. Usually the key (the line offset) is not passed to the mapper. The default command set explicitly is:

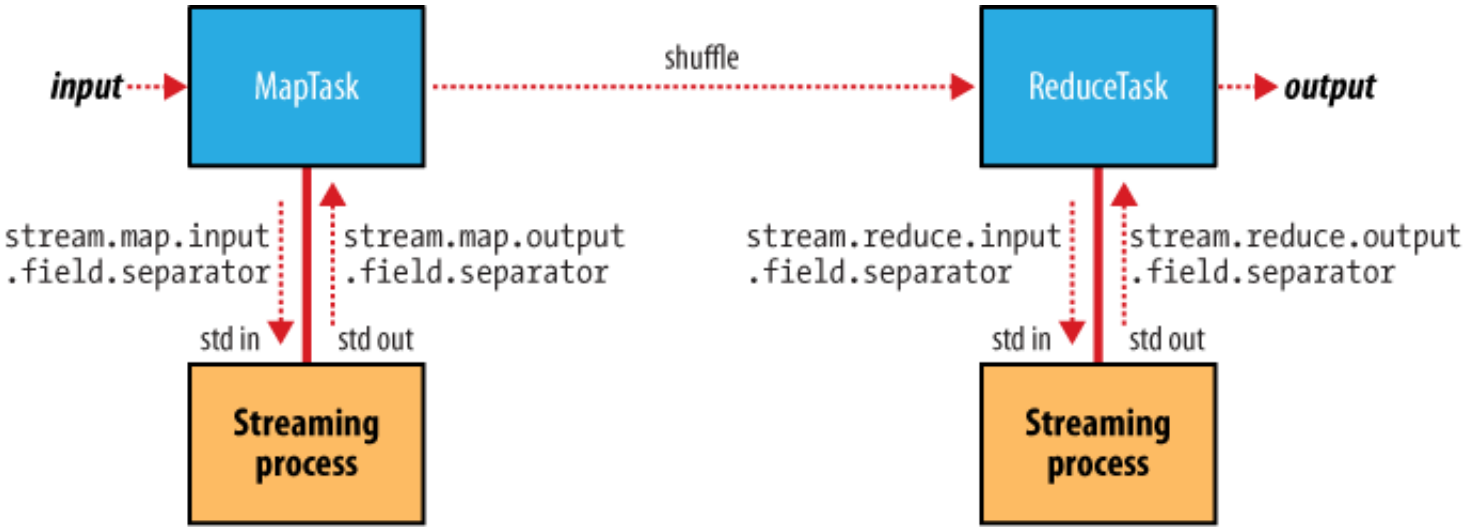
```
    $ hadoop jar $HADOOP_INSTALL/contrib/hadoop-*streaming.jar \
    -input input/sample.txt \
    -output output \
    - inputformat org.apache.hadoop.mapred.TextInputFormat \
    -mapper /bin/cat \
    -partitioner org.apache.hadoop.mapred.lib.HashPartitioner \
    -numReduceTasks 1 \
    -reducer org.apache.hadoop.mapred.lib.IdentityReducer \
    -outputformat org.apache.hadoop.mapred.TextOutputFormat
```

Keys and Values in Streaming

A Streaming app can control the separator used when a key-value pair is turned into a series of bytes (defaults is tab char)

Table 7-3. Streaming separator properties

Property name	Type	Defaultvalue	Description
`stream.map.input.field.separator`	`String`	`\t`	The separator to use when passing the input key and value strings to the stream map process as a stream of bytes.
`stream.map.output.field.separator`	`String`	`\t`	The separator to use when splitting the output from the stream map process into key and value strings for the map output.
`stream.num.map.output.key.fields`	`int`	`1`	The number of fields separated by `stream.map.output.field.separator` to treat as the map output key.
`stream.reduce.input.field.separator`	`String`	`\t`	The separator to use when passing the input key and value strings to the stream reduce process as a stream of bytes.
`stream.reduce.output.field.separator`	`String`	`\t`	The separator to use when splitting the output from the stream reduce process into key and value strings for the final reduce output.
`stream.num.reduce.output.key.fields`	`int`	`1`	The number of fields separated by `stream.reduce.output.field.separator` to treat as the reduce output key.



Input formats

InputSplits and Records

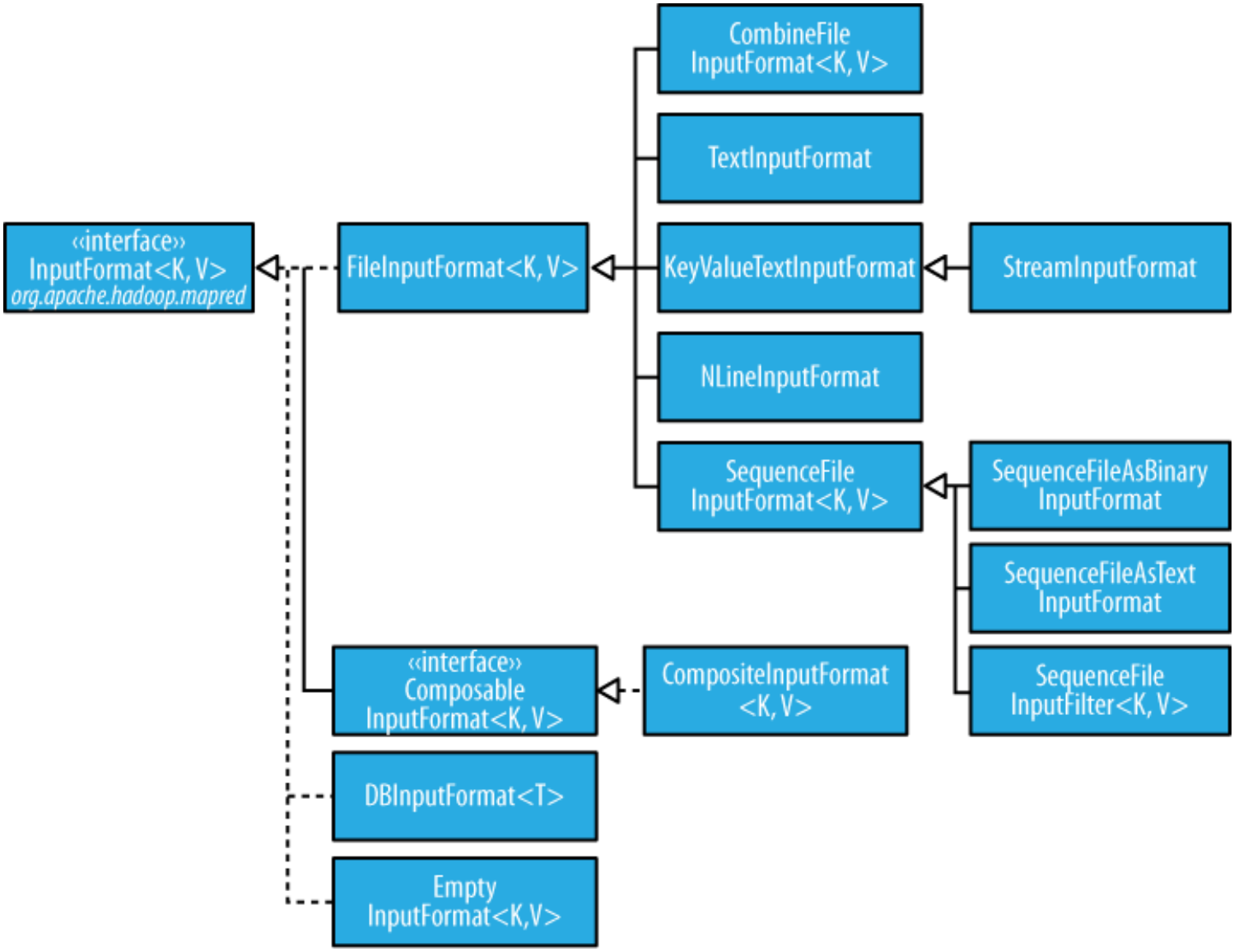
Each map processes a split that is divided into records. A client running a job calls **InputFormat.getSplits()** to retrieve the **InputSplit** list that is sent to the **JobTracker** to scheduling. On a **TaskTracker** the map passes the split to **InputFormat.createRecordReader()** to obtain the **RecordReader** (that is a iterator). The Map **run()** method:

```
    public void run(Context context) throws IOException, InterruptedException {
        setup(context);
        while (context.nextKeyValue()) {
            map(context.getCurrentKey(), context.   getCurrentValue(), context);
        }
        cleanup(context);
    }
```

run() method is public and may be customized.

FileInputFormat

Base class of **InputFormat** for files. The input is a collection of input paths (file, directory or both)



To exlude certain files from the input, you can use **setInputPathFilter()**, by default it excludes hidden files.

FileInputFormat input splits

FileInputFormat splits files over the HDFS block size. You could also set a min and max size.

Small files and CombineFileInputFormat

CombineFileInputFormat packs many small files into each split. It also takes into account the node and rack locality.

You could also use a **SequenceFile**: keys are filenames and values file contents.

Preventing splitting

Some apps don't want files to be split, to avoid it you can increase the min split size to be larger than the largest file or subclass **FileInputFormat** to override the **isSplittable()** to return false.

File information in the mapper

A mapper processign a file input split can find information about the split by calling the **getInputSplit()** on Mapper's **Context** object

Processing a whole file as a record

First, avoid file splitting and to have a **RecordReader** that delivers the file contents as the value of the record.

Text Input

TextInputFormat

Each record is a line of input. The key, a **LongWritable** is the byte offset (not line number) within the file of the beginning of the line.

KeyValueTextInputFormat

To interpret text files where the key needs to be something different than the line offset. For example, a file with key-values separated by commas:

```
a,2
a,3
b,1
```

You can specify the separator (the comma) via the **mapreduce.input.keyvaluelinerecordreader.key.value.separator**

NLineInputFormat

To send a fixed number of lines to the mapper use NLineInputFormat

XML

StreamXMLRecordReader is used, setting the input format to **StreamInputFormat** and the **stream.recordreader.class** property to **org.apache.hadoop.streaming.StreamXmlRecordReader**

Binary Input

SequenceFileInputFormat

The common way is to call **SequenceFileInputFormat** that will delivers the corresponding key-values, like IntWritable:Text but a casting is possible that it delivers only Text:Text with **SequenceFileAsTextInputFormat** or BytesWritable objects with **SequenceFileAsBinaryInputFormat**

Multiple Inputs

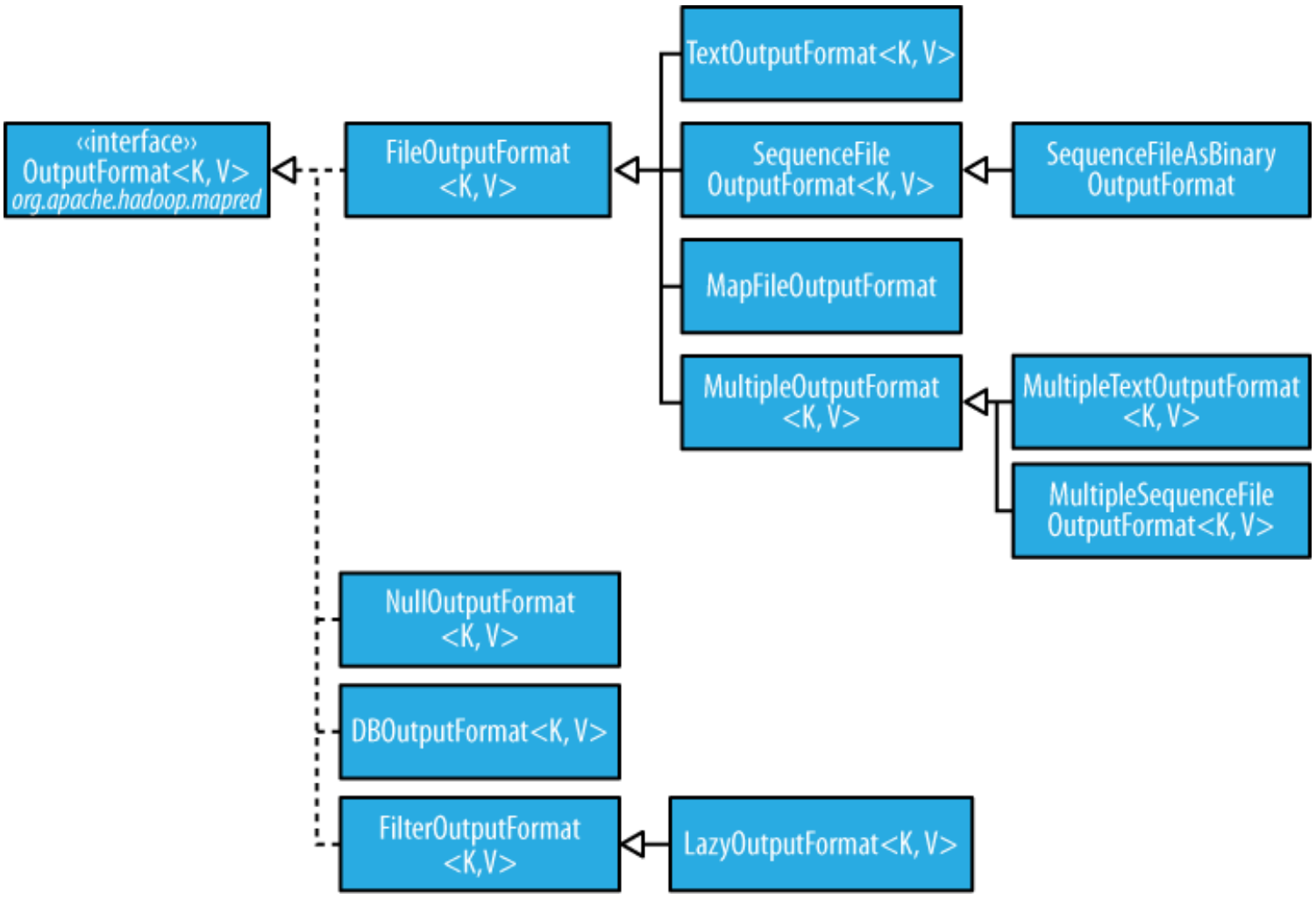
All of the inputs is interpreted by a single **InputFormat** and a single **Mapper**. But when you have different inputs you must use the **MultipleInputs** class that allows to use the **InputFormat** and **Mapper** on a per-path basis.

Database Input (and output)

DbInputFormat is for reading RDBS using JDBC. **DbOutputFormat** is for dumping the results to a DDBB.

TableInputFormat is used to read from **HBase**.

Output Formats



Text Output

The default **TextOutputFormat** writes records as lines of text tab-separated (that can be changed using `mapreduce.output.textoutputformat.separator`) calling to the **toString()** method on each record.

Binary Output

SequenceFileOutputFormat, SequenceFileAsBinaryOutputFormat and MapFileOutputFormat

The two classes are to create **SequenceFile** files, with and without compression. **MapFileOutputFormat** is to write MapFiles.

Multiple Outputs

For example, We will consult data from weather stations we would like to have a file for each station. To do this we need **MultipleOutputs** to write data to files whose names are derived from the output keys and values calling the **MultipleOutputs.write()** in the reducer instead of the Context.

LazyOutput

FileOutputFormat will create output (part-r-nnnnn) files even if they are empty. **LazyOutput** is used to avoid this.

MapReduce Features

Counters

Built-in Counters

Hadoop comes with some built-in counters to report various metrics for your job.

Table 8-1. Built-in counter groups

Group	Name/Enum
MapReduce Task Counters	`org.apache.hadoop.mapred.Task$Counter` (0.20) `org.apache.hadoop.mapreduce.TaskCounter` (post 0.20)
Filesystem Counters	`FileSystemCounters` (0.20) `org.apache.hadoop.mapreduce.FileSystemCounter` (post 0.20)
FileInputFormat Counters	`org.apache.hadoop.mapred.FileInputFormat$Counter` (0.20) `org.apache.hadoop.mapreduce.lib.input.FileInputFormatCounter` (post 0.20)
FileOutputFormat Counters	`org.apache.hadoop.mapred.FileOutputFormat$Counter` (0.20) `org.apache.hadoop.mapreduce.lib.output.FileOutputFormatCounter` (post 0.20)
Job Counters	`org.apache.hadoop.mapred.JobInProgress$Counter` (0.20) `org.apache.hadoop.mapreduce.JobCounter` (post 0.20)

Task Counters

Gather information about tasks over their execution and are maintained by each task attempt. Periodically they are sent to the jobtracker or, if YARN is used, they are fully sent.

Table 8-2. Built-in MapReduce task counters

Counter	Description
Map input records (MAP_INPUT_RECORDS)	The number of input records consumed by all the maps in the job. Incremented every time a record is read from a `RecordReader` and passed to the map's `map()` method by the framework.
Map skipped records (MAP_SKIPPED_RECORDS)	The number of input records skipped by all the maps in the job. See "Skipping Bad Records" on page 217.
Map input bytes (MAP_INPUT_BYTES)	The number of bytes of uncompressed input consumed by all the maps in the job. Incremented every time a record is read from a `RecordReader` and passed to the map's `map()` method by the framework.

Split raw bytes (SPLIT_RAW_BYTES)	The number of bytes of input split objects read by maps. These objects represent the split metadata (that is, the offset and length within a file) rather than the split data itself, so the total size should be small.
Map output records (MAP_OUTPUT_RECORDS)	The number of map output records produced by all the maps in the job. Incremented every time the `collect()` method is called on a map's `OutputCollector`.
Map output bytes (MAP_OUTPUT_BYTES)	The number of bytes of uncompressed output produced by all the maps in the job. Incremented every time the `collect()` method is called on a map's `OutputCollector`.
Map output materialized bytes (MAP_OUTPUT_MATERIALIZED_BYTES)	The number of bytes of map output actually written to disk. If map output compression is enabled this is reflected in the counter value.
Combine input records (COMBINE_INPUT_RECORDS)	The number of input records consumed by all the combiners (if any) in the job. Incremented every time a value is read from the combiner's iterator over values. Note that this count is the number of values consumed by the combiner, not the number of distinct key groups (which would not be a useful metric, since there is not necessarily one group per key for a combiner; see "Combiner Functions" on page 34, and also "Shuffle and Sort" on page 205).
Combine output records (COMBINE_OUTPUT_RECORDS)	The number of output records produced by all the combiners (if any) in the job. Incremented every time the `collect()` method is called on a combiner's `OutputCollector`.
Reduce input groups (REDUCE_INPUT_GROUPS)	The number of distinct key groups consumed by all the reducers in the job. Incremented every time the reducer's `reduce()` method is called by the framework.
Reduce input records (REDUCE_INPUT_RECORDS)	The number of input records consumed by all the reducers in the job. Incremented every time a value is read from the reducer's iterator over values. If reducers consume all of their inputs, this count should be the same as the count for Map output records.
Reduce output records (REDUCE_OUTPUT_RECORDS)	The number of reduce output records produced by all the maps in the job. Incremented every time the `collect()` method is called on a reducer's `OutputCollector`.
Reduce skipped groups (REDUCE_SKIPPED_GROUPS)	The number of distinct key groups skipped by all the reducers in the job. See "Skipping Bad Records" on page 217.
Reduce skipped records (REDUCE_SKIPPED_RECORDS)	The number of input records skipped by all the reducers in the job.
Reduce shuffle bytes (REDUCE_SHUFFLE_BYTES)	The number of bytes of map output copied by the shuffle to reducers.
Spilled records (SPILLED_RECORDS)	The number of records spilled to disk in all map and reduce tasks in the job.
CPU milliseconds (CPU_MILLISECONDS)	The cumulative CPU time for a task in milliseconds, as reported by */proc/cpuinfo*.
Physical memory bytes (PHYSICAL_MEMORY_BYTES)	The physical memory being used by a task in bytes, as reported by */proc/meminfo*.
Virtual memory bytes (VIRTUAL_MEMORY_BYTES)	The virtual memory being used by a task in bytes, as reported by */proc/meminfo*.
Committed heap bytes (COMMITTED_HEAP_BYTES)	The total amount of memory available in the JVM in bytes, as reported by `Runtime.getRuntime().totalMemory()`.
GC time milliseconds (GC_TIME_MILLIS)	The elapsed time for garbage collection in tasks in milliseconds, as reported by `GarbageCollectorMXBean.getCollectionTime()`. From 0.21.
Shuffled maps (SHUFFLED_MAPS)	The number of map output files transferred to reducers by the shuffle (see "Shuffle and Sort" on page 205). From 0.21.
Failed shuffle (FAILED_SHUFFLE)	The number of map output copy failures during the shuffle. From 0.21.

(FAILED_SHUFFLE)	
Merged map outputs (MERGED_MAP_OUTPUTS)	The number of map outputs that have been merged on the reduce side of the shuffle. From 0.21.

Table 8-3. Built-in filesystem task counters

Counter	Description
Filesystem bytes read (BYTES_READ)	The number of bytes read by each filesystem by map and reduce tasks. There is a counter for each filesystem: *Filesystem* may be Local, HDFS, S3, KFS, etc.
Filesystem bytes written (BYTES_WRITTEN)	The number of bytes written by each filesystem by map and reduce tasks.

Table 8-4. Built-in FileInputFormat task counters

Counter	Description
Bytes read (BYTES_READ)	The number of bytes read by map tasks via the FileInputFormat.

Table 8-5. Built-in FileOutputFormat task counters

Counter	Description
Bytes written (BYTES_WRITTEN)	The number of bytes written by map tasks (for map-only jobs) or reduce tasks via the FileOutputFormat.

Job Counters

Counter	Description
Launched map tasks (TOTAL_LAUNCHED_MAPS)	The number of map tasks that were launched. Includes tasks that were started speculatively.
Launched reduce tasks (TOTAL_LAUNCHED_REDUCES)	The number of reduce tasks that were launched. Includes tasks that were started speculatively.
Launched uber tasks (TOTAL_LAUNCHED_UBERTASKS)	The number of uber tasks (see "YARN (MapReduce 2)" on page 194) that were launched. From 0.23.
Maps in uber tasks (NUM_UBER_SUBMAPS)	The number of maps in uber tasks. From 0.23.
Reduces in uber tasks (NUM_UBER_SUBREDUCES)	The number of reduces in uber tasks. From 0.23.
Failed map tasks (NUM_FAILED_MAPS)	The number of map tasks that failed. See "Task Failure" on page 200 for potential causes.
Failed reduce tasks (NUM_FAILED_REDUCES)	The number of reduce tasks that failed.
Failed uber tasks (NUM_FAILED_UBERTASKS)	The number of uber tasks that failed. From 0.23.
Data-local map tasks (DATA_LOCAL_MAPS)	The number of map tasks that ran on the same node as their input data.
Rack-local map tasks (RACK_LOCAL_MAPS)	The number of map tasks that ran on a node in the same rack as their input data, but that are not data-local.
Other local map tasks (OTHER_LOCAL_MAPS)	The number of map tasks that ran on a node in a different rack to their input data. Inter-rack bandwidth is scarce, and Hadoop tries to place map tasks close to their input data, so this count should be low. See Figure 2-2.
Total time in map tasks (SLOTS_MILLIS_MAPS)	The total time taken running map tasks in milliseconds. Includes tasks that were started speculatively.
Total time in reduce tasks (SLOTS_MILLIS_REDUCES)	The total time taken running reduce tasks in milliseconds. Includes tasks that were started speculatively.
Total time in map tasks waiting after reserving slots (FALLOW_SLOTS_MILLIS_MAPS)	The total time spent waiting after reserving slots for map tasks in milliseconds. Slot reservation is Capacity Scheduler feature for high-memory jobs, see "Task memory limits" on page 316. Not used by YARN-based MapReduce.
Total time in reduce tasks waiting after reserving slots (FALLOW_SLOTS_MILLIS_REDUCES)	The total time spent waiting after reserving slots for reduce tasks in milliseconds. Slot reservation is Capacity Scheduler feature for high-memory jobs, see "Task memory limits" on page 316. Not used by YARN-based MapReduce.

User Defined Java Counters

Defined by a **Enum** and using a **Reporter.incrCounter()** object that can increment the counters.

Readable counter names

By default, a counter's name is the enum's fully qualified Java classname. To change it, you must create a properties file that must contain a single property named **CounterGroupName** with the value to display.

The file must be named using an underscore as separator for nested classes and it must be placed in the same directory as the top-level class containing the *enum*.

Retrieving counters

Using **hadoop job -counter** you can retrieve counter values.

User-Defined Streaming Counters

A Streaming program can also increment a counter by sending a formatted line to the standard error stream with the following format:

```
reporter:counter:group,counter,amount
```

In Python:

```
    sys.stderr.write("reporter:counter:Temperature,Missing,1\n")

## Sorting
To sort, we **must** sort a numeric key, but with **Text** it doesn't work

### Total sort: Producing a globally sorted file
Using a single partition is inneficient. The way is to produce a set of sorted files that concaten

| Temperature           | -10   | -10-0 | 0-10 | +10 |
| Proportion of records | 11%   | 13%   | 17%  | 59% |

This are not even. *Sampling* the key space to look at a small subset of the keys to approximate t

| Temperature           | -5.6  | -5.6-13.9 | 13.9-22 | +22 |
| Proportion of records | 29%   | 24%   | 23%  | 24% |

### Secondary sort
For any particular reducer key, the values are *not* sorted. To sort the values we must change the

For example the year and the temperature, that we want to sort by year and then by temperature. Ne

* Make the key a composite of key and value
* The sort comparator should order by the composite key, that is, the key and value
* The partitioner and grouping comparator for the composite key should consider only the key for p

## Joins
###Map-Side Joins
Performs the join before the data reaches the map function. Each input dataset must be divided int

![Inner join of two datasets](img/image-41.png)
```

```
Use a **CompositeInputFormat**to run a map-side join.

### Reduce-Side Joins
Is less efficient that the map-side. The mapper tags each record with its source and uses the join

* *Multiple inputs*: **MultipleInputs** must be used as the input sources are usually different
* *Secondary sort*: To perform a join it's important to have the data ordered.

## Side Data Distribution
Considered extra read-only data needed by a job to process the main dataset.

### Using the Job Configuration
Arbitrary key/value pairs can be set in the job configuration using the setters for **Configuratio

### Distributed Cache
#### Usage
For tools thatuse the **GenericOptionsParser* you can specify the files to be distributed as a com

Files will be localized under the **${mapred.local.dir}/taskTracker/archive within each TaskTracke

#### The distributed cache API
Putting data with **Job.addCacheXXXX()** and getting with **JobContext.setCacheXXXX()**.

![Cache API](img/image-42.png)

## MapReduce Library Classes
Hadoop comes with a library of mappers and reducers for commonly used functions:

![MapReduce Library classes](img/image-43.png)
```

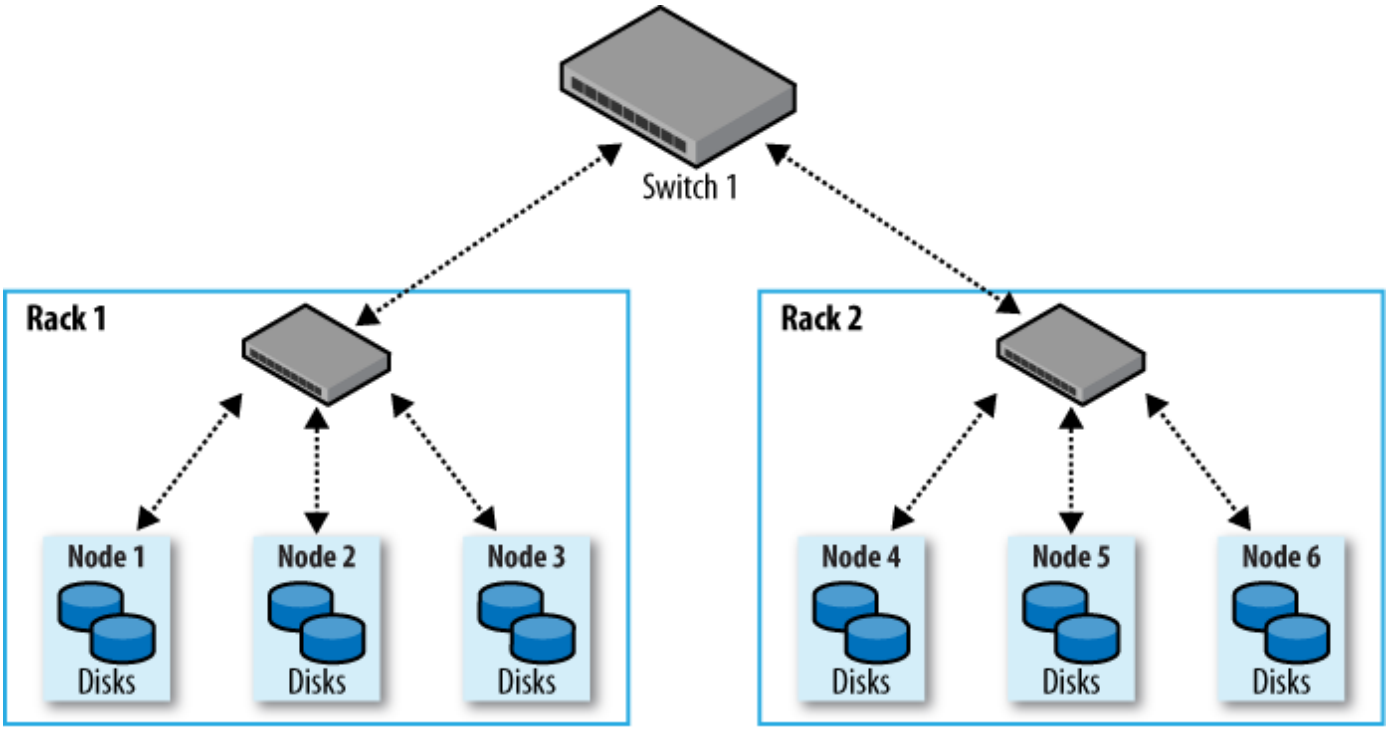
Setting up a Hadoop Cluster

Cluster specification

Hadoop uses "commodity hardware" that is easily available hardware (not low end). RAID is, however, not used for it's redundancy.

Network Topology

Typically 30 servers per rack with 1 gb switch.



Rack awareness

If more than one rack is used, Hadoop must be configured to know the topology of the network (that is presented as a tree).

Hadoop conf must specify a map between node addresses and network locations described by the interface:

```
    public interface DNSToSwitchMapping {
        public List<String> resolve(List<String> names);
    }
```

The **names** is a list of IP addresses and the return is a list of corresponding network location strings. However most installations don't need to implement the interface since the default implementation is **ScriptBasedMapping** which is located in **topology.script.file.name**.

Cluster Setup and Installation

Java 6 or later is needed. A Hadoop use is also a good practice and ssh password-less access between machines.

Table 9-1. Hadoop configuration files

Filename	Format	Description
hadoop-env.sh	Bash script	Environment variables that are used in the scripts to run Hadoop.
core-site.xml	Hadoop configuration XML	Configuration settings for Hadoop Core, such as I/O settings that are common to HDFS and MapReduce.
hdfs-site.xml	Hadoop configuration XML	Configuration settings for HDFS daemons: the namenode, the secondary namenode, and the datanodes.
mapred-site.xml	Hadoop configuration XML	Configuration settings for MapReduce daemons: the jobtracker, and the tasktrackers.
masters	Plain text	A list of machines (one per line) that each run a secondary namenode.
slaves	Plain text	A list of machines (one per line) that each run a datanode and a tasktracker.
hadoop-metrics.properties	Java Properties	Properties for controlling how metrics are published in Hadoop (see "Metrics" on page 350).
log4j.properties	Java Properties	Properties for system logfiles, the namenode audit log, and the task log for the tasktracker child process ("Hadoop Logs" on page 173).

Configuration Management

Hadoop does not have a single file, global located, for conf info. Each node has its own set of conf files and the admins must sync them.

You'll need a *class* of machines because not all machines will have the same hardware nor the same configuration files.

Control scripts

Hadoop slaves conf file can be anywhere setting the **HADOOP_SLAVES** in the *hadoop-env.sh*. Also **these files doesn't need to be distributed to worker nodes**.

You don't need to specify which machine the namenode and jobtracker runs on in the *masters* file as this is determined by the machine the scripts are run on.

Scripts:

- **start-dfs.sh**:

    - Starts a namenode on the local machine.
    - Starts a datanode in each *slave* file.
    - Starts a secondarynamenode in the *masters* file.

- **start-mapred.sh**:

    - Starts a Jobtracker on the local machine
    - Starts a TaskTracker on each machine listed in the slaves file.

Master node scenarios

With more than 10 nodes it's convenient to put the Jobtracker, namenodes and secondarynamenodes on different machines. Namenode has high memory requirements, also the secondarynamenode (when not idle) because it keeps a copy of the latest checkpoint of the filesystem metadata. When the master daemons run on one or more nodes:

- Run HDFS control scripts from the namenode
- Run the MR control scripts from the JobTracker

Environment Settings

Memory

Hadoop alllocates 1gb of memory to each daemon it runs (**HADOOP_HEAPSIZE** in *hadoop-env.sh*). In addition the tasktracker also launches separate child JVMs to run map and reduce tasks. The memory given to each JVM is `-Xmx200m` or 200mb

The max number of map/reduce tasks are set in `mapred.tasktracker.map/reduce.tasks.maximum` (default to 2). They are also related to number of processors available in a 2:1 ratio (two processes per processor). For example, with 8 processors if you want 2 processes on each, the maximum map/reduce tasks could be 7 (because datanode and tasktracker each take one slot). With higher (`-Xmx400m`) the total memory usage would be 7600mb that will run or not, depending of the rest of the processes (like if you use Streaming or not).

Master node, each of the namenode, secondary namenode and jobtracker needs 1gb each one.

System logfiles

Logfiles are store by default on **$HADOOP_INSTALL/logs** or in **HADOOP_LOG_DIR** within `hadoop-env.sh` file. The log4j is stored in a .log file and the combined standard output and error log and only the last five logs are retained.

Logfile names are a combination of the user, the daemon, the daemon name and the machine hostname.

SSH Settings

ConnectTimeout can be used to avoid that control scripts hang waiting a response. **StrictHostKeyChecking** can be set to **no** to automatically add new host keys to the known hosts files.

To pass extra options to SSH, define the **HADOOP_SSH_OPTS** in `hadoop-env.sh`.

Hadopo control scripts **can distribute config files to all nodes** of the cluster using *rsync* but isn't enabled by default. To enable, a **HADOOP_MASTER** must be defined in the `hadoop-env.sh` to point the directory to *rsync* to the nodes. The HADOOP_MASTER directory will be sync to the **HADOOP_INSTAL** dir.

Important Hadoop Daemon Properties

HDFS

Table 9-3. Important HDFS daemon properties

Property name	Type	Default value	Description
`fs.default.name`	URI	`file:///`	The default filesystem. The URI defines the hostname and port that the namenode's RPC server runs on. The default port is 8020. This property is set in *core-site.xml*.
`dfs.name.dir`	comma-separated directory names	`${hadoop.tmp.dir}/dfs/name`	The list of directories where the namenode stores its persistent metadata. The namenode stores a copy of the metadata in each directory in the list.
`dfs.data.dir`	comma-separated directory names	`${hadoop.tmp.dir}/dfs/data`	A list of directories where the datanode stores blocks. Each block is stored in only one of these directories.
`fs.checkpoint.dir`	comma-separated directory names	`${hadoop.tmp.dir}/dfs/namesecondary`	A list of directories where the secondary namenode stores checkpoints. It stores a copy of the checkpoint in each directory in the list.

MapReduce

Table 9-4. Important MapReduce daemon properties

Property name	Type	Default value	Description
`mapred.job.tracker`	hostname and port	`local`	The hostname and port that the jobtracker's RPC server runs on. If set to the default value of `local`, then the jobtracker is run in-process on demand when you run a MapReduce job (you don't need to start the jobtracker in this case, and in fact you will get an error if you try to start it in this mode).
`mapred.local.dir`	comma-separated directory names	`${hadoop.tmp.dir}/mapred/local`	A list of directories where MapReduce stores intermediate data for jobs. The data is cleared out when the job ends.
`mapred.system.dir`	URI	`${hadoop.tmp.dir}/mapred/system`	The directory relative to `fs.default.name` where shared files are stored, during a job run.
`mapred.tasktracker.map.tasks.maximum`	`int`	`2`	The number of map tasks that may be run on a tasktracker at any one time.
`mapred.tasktracker.reduce.tasks.maximum`	`int`	`2`	The number of reduce tasks that may be run on a tasktracker at any one time.
`mapred.child.java.opts`	`String`	`-Xmx200m`	The JVM options used to launch the tasktracker child process that runs map and reduce tasks. This property can be set on a per-job basis, which can be useful for setting JVM properties for debugging, for example.
`mapreduce.map.java.opts`	`String`	`-Xmx200m`	The JVM options used for the child process that runs map tasks. From 0.21.
`mapreduce.reduce.java.opts`	`String`	`-Xmx200m`	The JVM options used for the child process that runs reduce tasks. From 0.21.

Hadoop Daemon Addresses and Ports

Table 9-5. RPC server properties

Property name	Default value	Description
fs.default.name	file:///	When set to an HDFS URI, this property determines the namenode's RPC server address and port. The default port is 8020 if not specified.
dfs.datanode.ipc.address	0.0.0.0:50020	The datanode's RPC server address and port.
mapred.job.tracker	local	When set to a hostname and port, this property specifies the jobtracker's RPC server address and port. A commonly used port is 8021.
mapred.task.tracker.report.address	127.0.0.1:0	The tasktracker's RPC server address and port. This is used by the tasktracker's child JVM to communicate with the tasktracker. Using any free port is acceptable in this case, as the server only binds to the loopback address. You should change this setting only if the machine has no loopback address.

Table 9-6. HTTP server properties

Property name	Default value	Description
mapred.job.tracker.http.address	0.0.0.0:50030	The jobtracker's HTTP server address and port.
mapred.task.tracker.http.address	0.0.0.0:50060	The tasktracker's HTTP server address and port.
dfs.http.address	0.0.0.0:50070	The namenode's HTTP server address and port.
dfs.datanode.http.address	0.0.0.0:50075	The datanode's HTTP server address and port.
dfs.secondary.http.address	0.0.0.0:50090	The secondary namenode's HTTP server address and port.

Other Hadoop Properties

Cluster membership

A list of authorized machines are specified in the **dfs.hosts** for datanodes and **mapred.hosts** for tasktrackers as well as **dfs.hosts.exclude** and **mapred.hosts.exclude**.

Buffer size

Hadoop uses a 4kb buffer size for its I/O operations which is conservative and increasing to 128kb will give an improved performance in **io.file.buffer.size** in *core-site.xml*.

Reserved storage space

To reserve some space on the datanodes set **dfs.datanode.du.reserved** to the amount of bytes to reserve.

Trash

Minimum period a deleted file in HDFS is retain until ultimate deletion (**fs.trash.interval** defaults to 0 which is disable)

Reduce Slow Start

Reducers wait until 5% of the map tasks in a jab have completed before scheduling reduce tasks. For large jobs this will take slots while waiting. **mapred.reduce.slowstart.completed.maps** can be set to a higher value (like 0.8 = 80%)

Task memory limits

With *ulimit* or **mapred.child.ulimit** a limit for tasks is set

YARN Configuration

- **yarn-env.sh**: Environment variables
- **yarn-site.xml**: Configuration settings for YARN daemons: resourcemanager, jobhistory, webapp and node managers.

Important YARN Daemon Properties

mapred-site.xml is still used for general MapReduce properties.

Table 9-9. Important YARN daemon properties

Property name	Type	Default value	Description
`yarn.resourceman ager.address`	hostname and port	`0.0.0.0:8040`	The hostname and port that the resource manager's RPC server runs on.
`yarn.nodeman ager.local-dirs`	comma-separated directory names	`/tmp/nm-local-dir`	A list of directories where node managers allow containers to store intermediate data. The data is cleared out when the application ends.
`yarn.nodeman ager.aux-services`	comma-separated service names		A list of auxiliary services run by the node manager. A service is implemented by the class defined by the property `yarn.nodemanager.aux-services.`*`service-name`*`.class`. By default no auxiliary services are specified.
`yarn.nodeman ager.resource.mem ory-mb`	`int`	`8192`	The amount of physical memory (in MB) which may be allocated to containers being run by the node manager.

Property name	Type	Default value	Description
`yarn.nodeman ager.vmem-pmem-ratio`	`float`	`2.1`	The ratio of virtual to physical memory for containers. Virtual memory usage may exceed the allocation by this amount.

Memory

YARN allows to request an arbitrary amount of memory (within limits) for a task instead of the fixed slot-way of Hadoop v1. **yarn.nodemanager.resource.memory-mb** can be set to a limit that can't be trespassed.

Also in map and reduce the physical memory limits can be set with **mapreduce.map/reduce.memory.mb**

Yarn Daemon Addresses and Ports

Table 9-10. YARN RPC server properties

Property name	Default value	Description
yarn.resourceman ager.address	0.0.0.0:8040	The resource manager's RPC server address and port. This is used by the client (typically outside the cluster) to communicate with the resource manager.
yarn.resourceman ager.admin.address	0.0.0.0:8141	The resource manager's admin RPC server address and port. This is used by the admin client (invoked with `yarn rmadmin`, typically run outside the cluster) to communicate with the resource manager.
yarn.resourceman ager.scheduler.address	0.0.0.0:8030	The resource manager scheduler's RPC server address and port. This is used by (in-cluster) application masters to communicate with the resource manager.
yarn.resourceman ager.resource-tracker.address	0.0.0.0:8025	The resource manager resource tracker's RPC server address and port. This is used by the (in-cluster) node managers to communicate with the resource manager.
yarn.nodeman ager.address	0.0.0.0:0	The node manager's RPC server address and port. This is used by (in-cluster) application masters to communicate with node managers.
yarn.nodemanager.local izer.address	0.0.0.0:4344	The node manager localizer's RPC server address and port.
mapreduce.jobhis tory.address	0.0.0.0:10020	The job history server's RPC server address and port. This is used by the client (typically outside the cluster) to query job history. This property is set in *mapred-site.xml*.

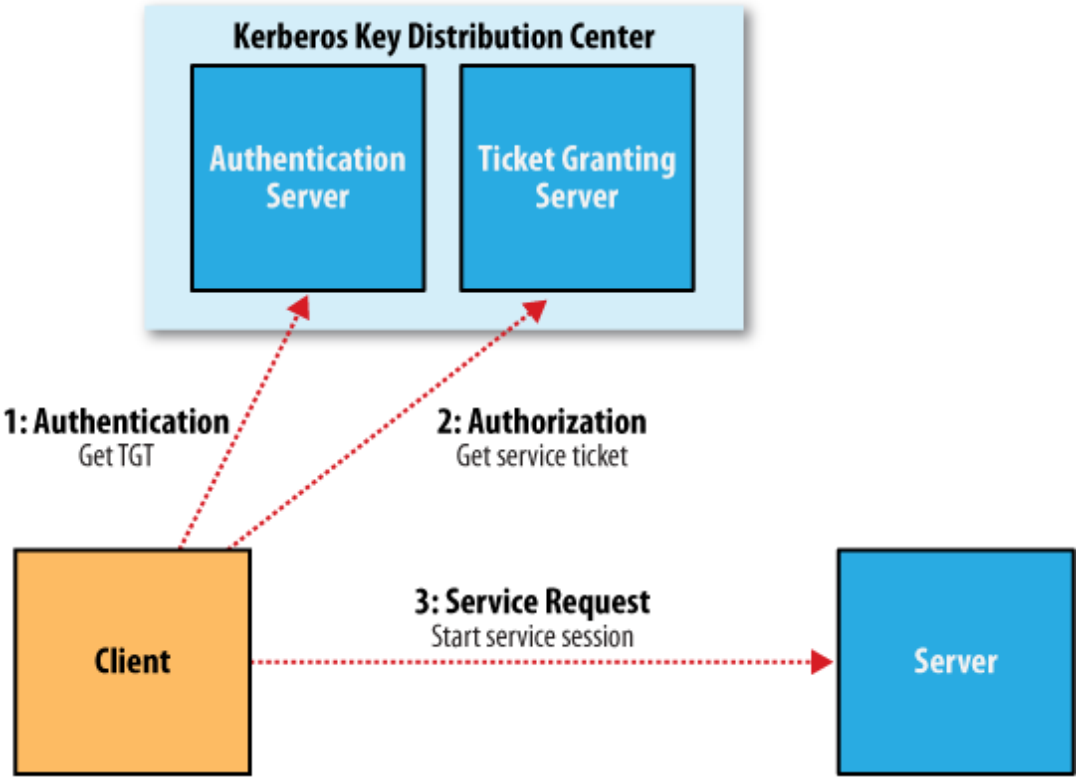
Table 9-11. YARN HTTP server properties

Property name	Default value	Description
yarn.resourceman ager.webapp.address	0.0.0.0:8088	The resource manager's HTTP server address and port.
yarn.nodeman ager.webapp.address	0.0.0.0:9999	The node manager's HTTP server address and port.
yarn.web-proxy.address		The web app proxy server's HTTP server address and port. If not set (the default) then the web app proxy server will run in the resource manager process.
mapreduce.jobhis tory.webapp.address	0.0.0.0:19888	The job history server's HTTP server address and port. This property is set in *mapred-site.xml*.
mapreduce.shuffle.port	8080	The shuffle handler's HTTP port number. This is used for serving map outputs, and is not a user-accessible web UI. This property is set in *mapred-site.xml*.

Security

Kerberos and Hadoop

1. *Authentication*: Client authenticates itself to the Authentication Server and receives a timestamped Ticket-Granting Ticket (TGT)
2. *Authorization*: The client uses the TGT to request a service ticket from the Ticket Granting Server
3. *Service Request*: The client uses the service ticket to authenticate itself to the server that is providing the service the client is using (namenode or jobtracker).



Kerberos is enabled with **hadoop.security.authentication** in *core-site.xml* and **hadoop.security.authorization** to true to enable service level auth. You may configure Access Control Lists (ACLs) in the *hadoop-policy.xml* file.

Delegation Token

Delegation tokes are used transparently by Hadoop, it is like a shared secret between the client and the server. They are generated by the server: it uses Kerberos on the first call and it will get a Delegation Token which the namenode can verify.

When it performs operations on HDFS, client uses a *block access token* that the namenode passes to the client in response to a metadata request. Client uses the block to access itself to datanodes. This closes the security hole where only the Job ID was needed to access to a block (by default enabled in **dfs.block.access.token.enable**)

When the job is finished, the delegation token are invalidated

Other Security Enhancements

- Tasks can be run using the OS user rather thatn the tasktracker's user (**mapred.task.tracker.task-controller** to **org.apache.hadoop.mapred.LinuxTaskController**)

- When tasks are run as the user who submitted the job, the distributed cache is secure.
- Users can view and modify only their own jobs (**mapred.acls.enabled** to **true**)
- Shuffle is secure but not encrypted
- It's no longer possible for a malicious user to run a rogue node that can join the cluster. Daemons must authenticate against the namenode
- A datanode may be run on a privileged port (under 1024) so a client may be reasonably sure that it was started securetly.
- A task may only communicate with its parent tasktracker.

Benchmarking a Hadoop Cluster

Hadoop Benchmarks

Benchmarking HDFS with TestDFSIO

TestDFSIO tests I/O performance of HDFS by reading or writing files. For example, to write 10 files of 1000 mb each:

```
$ hadoop jar $HADOOP_INSTALL/hadoop-*-test.jar TestDFSIO -write -nrFiles 10 -fileSize 1000
```

A read benchmark

```
$ hadoop jar $HADOOP_INSTALL/hadoop-*-test.jar TestDFSIO -read -nrFiles 10 -fileSize 1000
```

Benchmarking MapReduce with Sort

Good for testing because it sends a full input dataset through the shuffle.

1. Generate some random data: `hadoop jar $HADOOP_INSTALL/hadoop-*-examples.jar randomwriter random-data`
2. Run the sort: `hadoop jar $HADOOP_INSTALL/hadoop-*-examples.jar sort random-data sorted-data`
3. Validate:
   `hadoop jar $HADOOP_INSTALL/hadoop-*-test.jar testmapredsort -sortInput random-data -sortOutput sorted-data`

Other benchmarks

- **MRBench**: runs a small job a number of times
- **NNBench**: load testing namenode hardware
- **Gridmix**: suite of benchmarks to model a realistic cluster overload

Hadoop In The Cloud

Hadoop on Amazon EC2

Use **Whirr** telling it the credentials and for launching. An example:

```
bin/whirr launch-cluster --config recipes/hadoop-ec2.properties --private-key-file ~/.ssh/id_rsa_whirr
```

Configuration

Configuration is passed as bundles in a conf file with the **--config** option:

```
whirr.cluster-name=hadoop
whirr.instance-templates=1 hadoop-namenode+hadoop-jobtracker, 5 hadoop-datanode+hadoop-tasktracker
whirr.provider=aws-ec2
whirr.identity=${env:AWS_ACESS_KEY_ID}
whirr.credential=${env:AWS_SECRET_ACESS_KEY}
whirr.hardware-id=c1.xlarge
whirr.image-id=us-east-1/ami-da0cf8b3
whirr.location-id=us-east-1
```

All the options are self explanatory. The identity and credential uses environment variables set with `export` in bash. You can also override any of this properties passing them (without the whirr prefix) throught the command line.

Running a proxy

To set up the proxy:

```
$ . ~/.whirr/hadoop/hadoop-proxy.sh
```

Running a MapReduce Job

You can run a MR job from within the cluster or from an external machine.

Whirr conf files are in ~/.whirr/hadoop and can be used to connect to the cluster by setting the **HADOOP_CONF_DIR** in *hadoop-env.sh* to it.

Then we need to populate the cluster with data from, for example, S3:

```
$ hadoop distcp \
    -Dfs.s3n.awsAccessKeyId='...' \
    -Dfs.s3n.awsSecretAccessKey='...' \
    s3n://hadoopbook/ncdc/all input/ncdc/all
```

The the job is run as usual. To write the output to S3 the command is as follows:

```
$ hadoop jar hadoop-examples.jar MyJob /input s3n://mybucket/output
```

Shutting Down a Cluster

```
$ bin/whirr destroy-cluster --config recipes/hadoop-ec2.properties
```

Administering Hadoop

HDFS

Persistent Data Structures

Namenode Directory Structure

{dfs.name.dir}/current/VERSION /edits /fsimage /fstime

VERSION is a Java properties file that contains information about the version of HDFS: #Tue Mar 10 19:21:36 GMT 2009 namespaceID=134368441 cTime=0 storageType=NAME_NODE layoutVersion=-18

layoutVersion defines the version of HDFS persistent data structures. Whenever the layout changes, the version is decremented

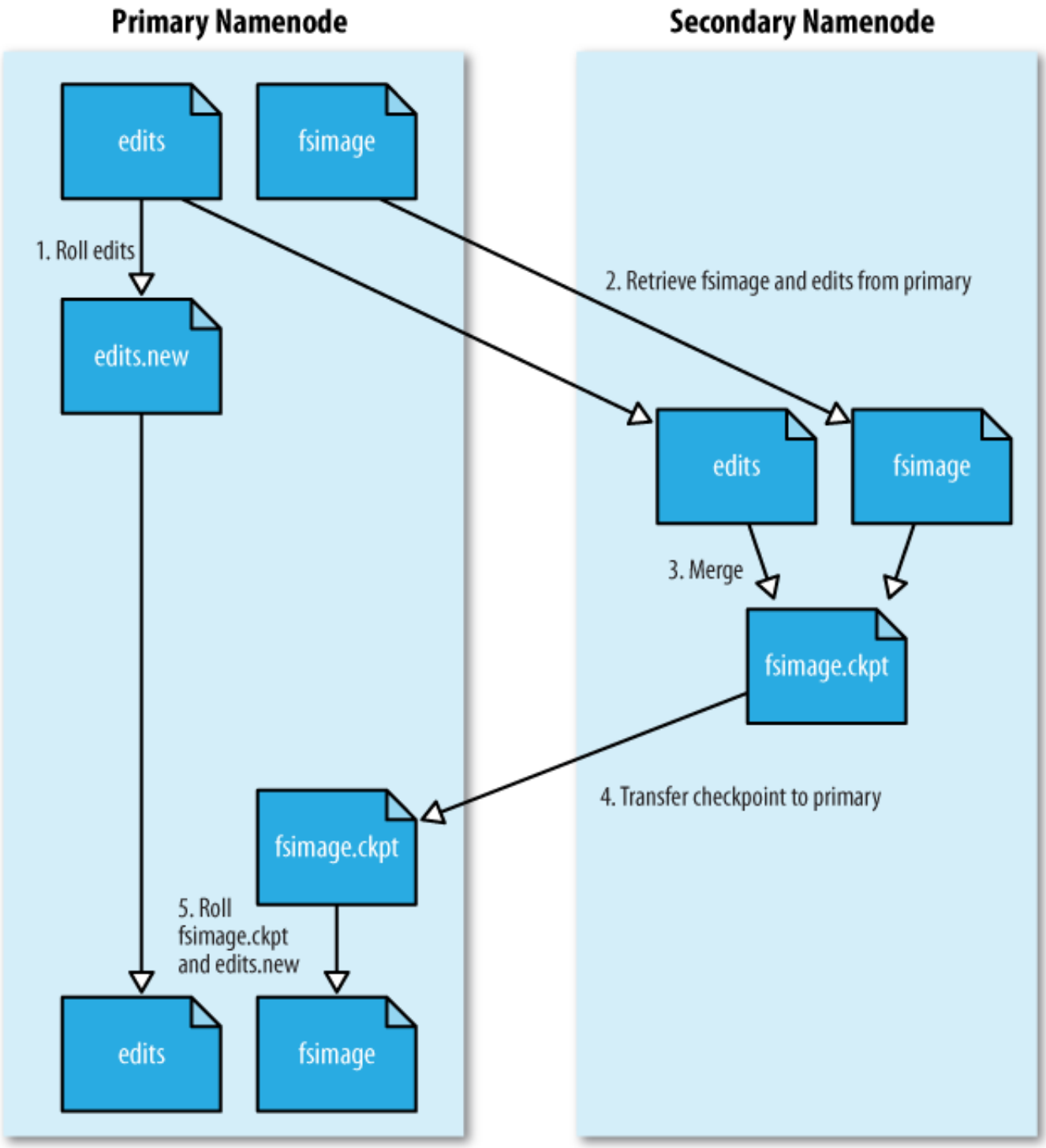
namespaceID Unique identifier for the filesystem to identify new datanodes.

cTime: marks the creation of the namenode's storage

storageType: Indicates that this storage directory contains data structures for a namenode.

The filesystem and edit log

When a filesystem client performs a write operation it is first recorded in the edit log. The *fsimage* file is a persisten checkpoint of the metadata.



Secondary namenode directory structure

Is identical to the namenode and it can be used to do manual backups.

Datanode directory structure

Has two types: HDFS blocks and the metadata for a block. When the blocks grows to 64 block new directory is created

Safe Mode

When the namenode starts, the first thing it does is load its *fsimage* into memory and apply the edits from the *edits*.

Is needed to give datanodes time to check in to the namenode with their block lists so the namenode can be informed of enough block locations to run the filesystem.

Entering and leaving safe mode

To check:

```
hadoop dfsadmin -safemode get
```

To wait for the namenode to exit safe mode:

```
hadoop dfsadmin -safemode wait
```

To enter safemode:

```
hadoop dfsadmin -safemode enter
```

To leave safemode:

```
hadoop dfsadmin -safemode leave
```

Audit Logging

HDFS can log filesystem access requests with *log4j*.

Tools

dfsadmin

Is a tool for finding information about the state of HDFS.

Filesystem check (fsck)

For checking health of files in HDFS. It looks for:

- Over replicated blocks

- Under replicated blocks
- Misreplicated blocks (those that do not satisfy the block replica placement)
- Corrupt blocks
- Missing replicas

Finding the blocks for a file hadoop fsck /part -files -blocks -racks

Datanode block scanner

Blocks are periodically verified (`dfs.datanode.scan.period.hours`) to check for corrupt ones by scanning.

Also, the distribution of blocks can become unbalanced. The **balancer** is a daemon that re-distributes blocks.

Monitoring

Logging

All daemons produce logfiles, to **set the log levels** it can be change from their web pages

Metrics

They belongs to a context: dfs, mapred, rpc and jvm. Are configured in the **conf/hadoop-metrics.properties** file, by default they do not publish:

```
dfs.class=org.apache.hadoop.metrics.spi.NullContext
mapred.class=org.apache.hadoop.metrics.spi.NullContext
jvm.class=org.apache.hadoop.metrics.spi.NullContext
rpc.class=org.apache.hadoop.metrics.spi.NullContext
```

- **FileContext** Writes metrics to a local file exposing two properties: **filename** and **period**
- **GangliaContext** Is a open-source distributed monitoring system. By using **GagliaContext** you can inject Hadoop metrics into Ganglia by, at least, defining one required property: **servers**.
- **NullContextWithUpdateThread** Pulls metrics from Hadoop by periodically updating metrics stored in memory.
- **CompositeContext** Output the same metrics to multiple context.

Java Management Extensions

!Hadoop MBeans

Nagios and Hyperic can query MBeans for monitoring. All options for enabling remote access to JMX involve setting Java system properties editing *conf/hadoop-env.sh*

Maintenance

Routine Administration Procedures

Metadata backups

If the namenode's persistent metadata is lost or damaged, the entire filesystem is rendered unusable so a backup can be done of the secondary's namenode *previous.checkpoint* property.

Data backups

The data to backup must be prioritized. Space quotas can be used to backup data at night.

Commissioning and Decommissioning Nodes

New nodes

Configure the *hdfs-site.xml* to point to the namenode and the *mapred-site.xml* to the jobtracker and launch datanode and jobtracker. You must have a list of authorized nodes: `dfs.hosts` for the namenode and `mapred.hosts` for the jobtracker.

1. Add the network addresses of the new nodes to the include file
2. Update the namenode with the new set of permitted datanodes: hadoop dfsadmin -refreshNodes
3. Update the jobtracker with the new set of permitted tasktrackers: hadoop mradmin -refreshNodes
4. Update the slaves file
5. Start the new datanodes and tasktrackers
6. Check that the new nodes appears in the web UI

Decommissioning nodes

If you shut down a tasktracker running tasks, they will be re-scheduled:

1. Add the network addresses of the nodes to the exclude file.
2. Update the namenode hadoop dfsadmin -refreshNodes

3. Update the jobtracker hadoop dfsadmin -refreshNodes
4. Go to the web UI and check whether the admin state has changed to "Decommission in progress"
5. When all datanodes report their state as "Decommissioned", then all the nodes have been replicated. Shut down the deccommissioned nodes.
6. Remove the nodes from the include file and run: hadoop dfsadmin -refreshNodes haddop mradmin -refreshNodes
7. Remove the nodes from the slaves file

Upgrades

1. Make sure that any previous upgrade is finalized before proceeding with another upgrade.
2. Shut down MapReduce and kill any orphaned task processes on the tasktrackers.
3. Shut down HDFS and backup the namenode directories.
4. Install new versions of Hadoop HDFS and MapReduce on the cluster and on clients.
5. Start HDFS with the -upgrade option.
6. Wait until the upgrade is complete.
7. Perform some sanity checks on HDFS.
8. Start MapReduce.
9. Roll back or finalize the upgrade (optional).

- Start the upgrade $NEW_HADOOP_INSTALL/bin/start-dfs.sh -upgrade
- Wait until the upgrade is complete $NEW_HADOOP_INSTALL/bin/hadoop dfsadmin -upgradeProgress status

Pig

- Pig latin (a language to express data flows)
- Execution environment: local or on hadoop cluster

Pig is for explore large datasets and is extensible. It cannot explore subsets.

Hadoop mode

Pig translates queries into MapReduce jobs. The pig.properties must have:

```
fs.default.name=hdfs://localhost/
mapred.job.tracker=localhost:8021
```

Running Pig Programs

- **Script**: A file with commands. PigPen is an Eclipse Plugin
- **Grunt**: Interactive shell
- **Embedded**: From Java using **PigServer** class

Some example commands

- **LOAD (sample.txt) as (year:chararray, temperature int);** loads a file delimited by tabs
- **DUMP var;** Shows the contents of a var
- **DESCRIBE var;** Shows info about a var
- **FILTER records BY temperature != 9999** Filters a loaded file
- **GROUP filtered_records BY year;** Groups the contents using a column
- **FOREACH grouped_records GENERATE group;** Foreach process every row to to generate a derived set of rows with the fields of the GENERATE clause.
- **ILLUSTRATE var;** Shows pretty info of a var.

Table 11-1. Pig Latin relational operators

Category	Operator	Description
Loading and storing	LOAD	Loads data from the filesystem or other storage into a relation
	STORE	Saves a relation to the filesystem or other storage
	DUMP	Prints a relation to the console
Filtering	FILTER	Removes unwanted rows from a relation
	DISTINCT	Removes duplicate rows from a relation
	FOREACH...GENERATE	Adds or removes fields from a relation
	MAPREDUCE	Runs a MapReduce job using a relation as input
	STREAM	Transforms a relation using an external program
	SAMPLE	Selects a random sample of a relation
Grouping and joining	JOIN	Joins two or more relations
	COGROUP	Groups the data in two or more relations
	GROUP	Groups the data in a single relation
	CROSS	Creates the cross-product of two or more relations
Sorting	ORDER	Sorts a relation by one or more fields
	LIMIT	Limits the size of a relation to a maximum number of tuples
Combining and splitting	UNION	Combines two or more relations into one

Pig Latin

Structure and statements

A Pig Latin script is a collection of statements. The interpreter checks that each statement is semantically and sintactically correct and adds it to the **logical plan** but does *not* execute the command yet until a **DUMP** or **STORE** statement is read.

With **REGISTER, DEFINE** and **IMPORT** you can register UDF and macros to Pig. There are also *commands* that are executed without being added to the logical plan like **set, rmf, exec** or **mkdir**. **set** is used to control Pig's behaviour througth options: `set debug on`.

Expressions

Category	Expressions	Description	Examples
Field by position	$n	Field in position *n* (zero based)	$0
Field by name	f	Field named f	year
Field (disambiguation)	r::f	Field named f from relation r after grouping	A::year
Projection	c.$n, c.f	Field in container c, by position or name	record.$0, record.year
Map lookup	m#k	Value associated with key k in map m	items#'Coat'
Cast	(t) f	Cast of field f to type t	(int) year
Conditional	x ? y : z	y if x evaluates to true, z if not	quality == 0 ? 0 : 1
Flatten	FLATTEN(f)	Removal of a level of nesting from bags and tuples	FLATTEN(group)

Types

Table 11-6. Pig Latin types

Category	Type	Description	Literal example
Numeric	int	32-bit signed integer	1
	long	64-bit signed integer	1L
	float	32-bit floating-point number	1.0F
	double	64-bit floating-point number	1.0
Text	chararray	Character array in UTF-16 format	'a'
Binary	bytearray	Byte array	Not supported
Complex	tuple	Sequence of fields of any type	(1,'pomegranate')
	bag	An unordered collection of tuples, possibly with duplicates	{(1,'pomegranate'),(2)}
	map	A set of key-value pairs. Keys must be character arrays; values may be any type	['a'#'pomegranate']

TOTUPLE, TOBAG and **TOMAP** are used to turn expressions into those types. A relation is a top-level construct whereas a bag has to be contained in a relation

Schemas

Using the **AS** you can attach a schema to a **LOAD** statement.

```
records = LOAD '/input' AS (temp:int, year:int)
```

Schema (type) declaration can be omitted too.

```
records = LOAD '/input' AS (temp, year)
```

Anyways, schema is optional:

```
records = LOAD '/input'
```

Validation and null

When trying to load a string into an int, it will fail and a null will be produce after a warning. **SPLIT** can also be used to split the data into *good* and *bad* data:

```
SPLIT records INTO good_records IF temperature is not null;
```

Functions

1. **Eval function**: A function takes one or more expressions and returns another expression.
2. **Filter function**: Returns a boolean result. Are used in FILTER operator.
3. **Load function**: Specifies how to load data into a relation from external storage.
4. **Store function**: A function that specifies how to save the contents of a relation to external storage.

Category	Function	Description
	DIFF	Calculates the set difference of two bags. If the two arguments are not bags, then returns a bag containing both if they are equal; otherwise, returns an empty bag.
	MAX	Calculates the maximum value of entries in a bag.
	MIN	Calculates the minimum value of entries in a bag.
	SIZE	Calculates the size of a type. The size of numeric types is always one; for character arrays, it is the number of characters; for byte arrays, the number of bytes; and for containers (tuple, bag, map), it is the number of entries.
	SUM	Calculates the sum of the values of entries in a bag.
	TOBAG	Converts one or more expressions to individual tuples which are then put in a bag.
	TOKENIZE	Tokenizes a character array into a bag of its constituent words.
	TOMAP	Converts an even number of expressions to a map of key-value pairs.
	TOP	Calculates the top *n* tuples in a bag.
	TOTUPLE	Converts one or more expressions to a tuple.
Filter	IsEmpty	Tests if a bag or map is empty.
Load/Store	PigStorage	Loads or stores relations using a field-delimited text format. Each line is broken into fields using a configurable field delimiter (defaults to a tab character) to be stored in the tuple's fields. It is the default storage when none is specified.
	BinStorage	Loads or stores relations from or to binary files. A Pig-specific format is used that uses Hadoop `Writable` objects.
	TextLoader	Loads relations from a plain-text format. Each line corresponds to a tuple whose single field is the line of text.
	JsonLoader, JsonStorage	Loads or stores relations from or to a (Pig-defined) JSON format. Each tuple is stored on one line.
	HBaseStorage	Loads or stores relations from or to HBase tables.

Macros

For example, we can extract the part of our Pig Latin program that performs grouping on a relation then finds the maximum value in each group, by defining a macro as follows:

```
DEFINE max_by_group(X, group_key, max_field) RETURNS Y {
    A = GROUP $X by $group_key;
    $Y = FOREACH A GENERATE group, MAX($X.$max_field);
};
```

User defined functions (UDF)

Filter UDF

Filter UDF's are all subclasses of **FilterFunc** which itself is a subclass of **EvalFunc** and implements the **exec(Tuple tuple)** methods. A **Tuple** is a list of type defined objects with **Tuple.get(int index)** to retrieve the objects within it. **exec** must return **True** or **False** if the row must be contained in the filter. The last step is to use **REGISTER** to use the function.

Leveragin types

When a "cell" has an invalid value (String casting to int for example), the UDF will fail. Error handling could be done within the function but it's better to tell Pig the types of the fields that the function expects with **getArgToFuncMapping()**

Eval UDF

Must subclass **EvalFunc** and implement **exec()** and return null or the *parsed* object.

Dynamic invokers

Allows to call Java object from Pig but they are done by reflection and with large datasets can impose overhead. For example, to use **StringUtils** class:

```
grunt> DEFINE trim InvokeForString('org.apache.commons.lang.StringUtils trim', 'String');
grunt> B = FOREACH a GENERATE trim(fruit);
grunt> DUMP B;
(pomegranate)
(banana)
(apple)
```

Load UDF

Must extend **LoadFunc** and override:

- **setLocation(String location, Job job)**: Pass the input path location.
- **getInputFormat()**: Creates a RecordReader
- **prepareToRead(RecordReader reader, PigSplit split)**: Takes previous RecordReader.

- **getNext()**: Iterates throught the records

When using a schema, the fields need converting to the relevant type overriding **LoadCaster.getLoadCaster()** to provide a collection of conversion methods

Data processing operators

Storing data

```
STORE a INTO 'out' USING PigStorage(":");
```

Filtering Data

FOREACH...GENERATE

Acts on every row in a relation. It's similar to AWK, can be nested and can be used with UDF's.

STREAM

Transforms data using an external script:

```
grunt> C = STREAM A THROUGHT 'cut -f 2';
```

You can provide a custom serializer implementing **PigToStream** and **StreamToPig**

Grouping and joining data

JOIN

```
C = JOIN A BY $0, B BY $1; --inner join
```

Fragment replicate join can be used when the relations can fit in memory:

```
grunt> C = JOIN a BY $0, b BY $1 USING "replicated";
```

COGROUP

Returns a nested set of output tuples generating a tuple for each unique key.

```
grunt> d = COGROUP a BY $0, b BY $1;
grunt> DUMP d;
( 0, {}, {(Ali, 0)} )
( 1, {(1, Scarf)}, {} )
( 2, {(2, Tie)}, {(2, Joe), (2, Hank)} )
```

CROSS

Joins every tuple in a relation with every tuple in a second relation (watch out, the result will be even bigger than the source):

```
grunt> DUMP x;
(a)
(b)
grunt> DUMP y;
(c)
(d)
grunt> z = CROSS x, y;
grunt> DUMP z;
(a, c)
(a, d)
(b, c)
(b, d)
```

GROUP

Groups the data into a single relation

```
grunt> DUMP a;
(Joe, cherry)
(Ali, apple)
(Joe, banana)
grunt> b = GROUP a BY SIZE($1);
grunt> DUMP b;
(5, {(Ali, apple)})
(6, {(Joe, cherry), (Joe, banana)})
```

Sorting data

```
grunt> b = ORDER a BY $0, $1 DESC;
```

But any relation made after doesn't guarantee the order.

Combining and splitting data

To combine several relations into one:

```
grunt> DUMP a;
(2,3)
(1,2)

grunt> DUMP b;
(z,x,8)
(w,y,1)

grunt> c = UNION a, b;
grunt> DUMP c;
(2,3)
(1,2)
(z,x,8)
(w,y,1)
```

Pig tries to merge the schemas from both relations if possible or leave it without schema if not.

Pig in Practice

Parallelism

Pig uses one reducer per 1gb of input up to a maximum of 999 reducers (`pig.exec.reducers.bytes.per.reducer` and `pig.exec.reducers.max`). **PARALLEL** and `set default_parallel` allows to set explicitly the number of reducers.

```
grunt> g = GROUP a BY year PARALLEL 30;
-- same as
grunt> set default_parallel 30
grunt> g = GROUP a BY year;
```

Parameter substitution

You can send parameters into a script file that are read at runtime. For example a script that performs some analysis per day could receive the current day as param. For example **$input** and **$output** would specify input and output paths:

```
--script
r = LOAD '$input' AS (year, temp)
-- dome some hack that creates z as result
STORE z INTO '$output'
```

And this will be called using the following bash command:

```
pig -param input=/user/input
    -param output=/user/output
    file.pig
```

Hive

Running scripts

From the shell, simply run:

```
$ hive -f script.q

# Or

$ hive -e 'SELECT * FROM dummy'
```

- http://reference.jumpingmonkey.org/database/mysql/cheatsheet.html
- http://hortonworks.com/blog/hive-cheat-sheet-for-sql-users/

Commands

Creating tables

```
CREATE TABLE records (year STRING, temp INT, quality INT)
    ROW FORMAT DELIMITED
        FIELDS TERMINATED BY '\t';
```

Lines:

1. Create a table named "records" with 3 columns
2. Each row is tab delimited

Populating tables

```
LOAD DATA LOCAL INPATH 'input/sample.txt'
    OVERWRITE INTO TABLE records;
```

- Put the local file in the warehouse (no parsing)

`fs.default.name` is set to the default value of `file:///` so it will save to local. **Tables are stored as directories** under Hive's warehouse controller by `hive.metastore.warehouse.dir` property that defaults to */user/hive/warehouse*

OVERWRITE in the LOAD DATA tells Hive to delete any existing files.

Running Hive

Configuring Hive

In the `hive-site.xml` file you can set the typical `fs.default.name` and `mapred.job.tracker` to point the processes. It can also set per-session properties using `-hiveconf` to the hive command like this:

```
hive -hiveconf fs.default.name=localhost
```

During a session it's possible to use `SET` to change any property. The priority list is the following:

1. SET
2. -hiveconf
3. hive-site.xml
4. hive-default.xml
5. hadoop-site.xml
6. hadoop-default.xml

Logging

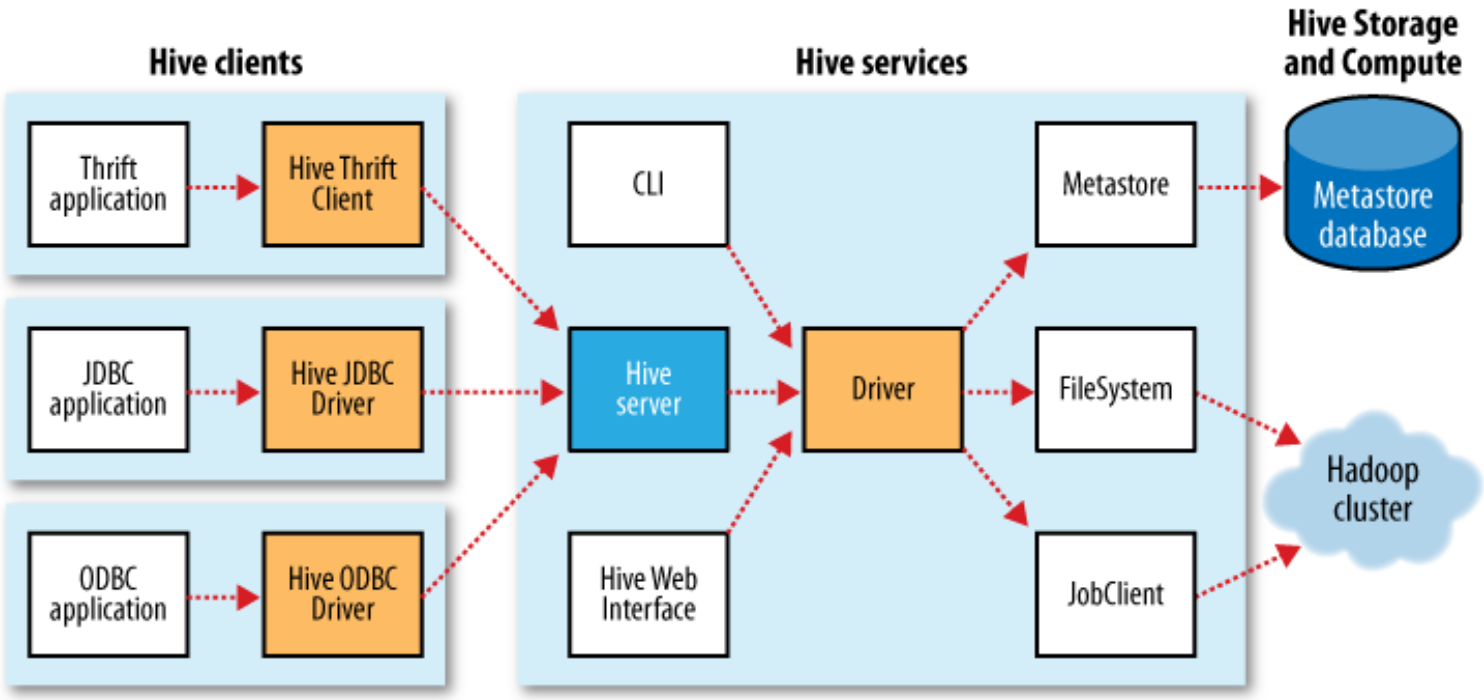
Found in /tmp/$USER/hive.log. The log conf file is in `conf/hive-log4j.properties`

Hive Services

Using `$ hive --service` option you can run the following:

1. cli: command line interface
2. hiserver: Hive server exposing a Thrift service for consumption (life JDBC)
3. hwi: Hive Web Interface
4. jar: Hive equivalent to `hadoop jar`
5. metastore: Used to run metastore as standalone

Hice Clients



1. Thrift Client: Makes easy to run Hive commands from many programming languages.
2. JDBC Driver: `org.apache.hadoop.hive.jdbc.HiveDriver` using a form like `jdbc:hive://host:port/dbname`
3. ODBC Driver: Support for ODBC protocol

The Metastore

A service and a backing store running on the same JVM (embedded metastore) that gives one session to Hive. With standalone database you can use multiple sessions (users)

The standalone metastore can be MySQL. The service must set to use the remote metastore with `hive.meta.store.local` to **false** and `hive.metastore.uris` to the metastore URI:

Property name	Type	Default value	
hive.metastore.warehouse.dir	URI	/user/hive/warehouse	D re fs w m

			ta st
hive.metastore.local	boolean	true	W a m re re ne
hive.metastore.uris	comma-separated-URI's	Not set	U sp re m
javax.jdo.option.ConnectionURL	String	jdbc:derby:;db=metastore_db;create=true	JI th da
javax.jdo.option.ConnectionDriverName	String	org.apache.derby.jdbc.EmbeddedDriver	JI cl
javax.jdo.option.ConnectionUserName	String	APP	JI na
javax.jdo.option.ConnectionPassword	String	mine	JI P

Comparison with Traditional Databases

Schema on Read Versus Schema on Write

In traditional the schema is enforced at data load time and if the data doesn't fit is rejected (*schema on write*). Hive verify in the query (*schema on read*).

HiveQL

https://cwiki.apache.org/confluence/display/Hive/LanguageManual

(Hortonworks.CheatSheet.SQLtoHive.pdf)

Data types

Primitives

Type	Description	Examples
TINYINT	1-byte signed int	1
SMALLINT	2-byte signed int	1
INT	4-byte signed int	1
BIGINT	8-byte signed int	1
FLOAT	4-byte floating point	1.0
DOUBLE	8-byte floating point	1.0
BOOLEAN	true/false value	TRUE
STRING	Character String	'a', "a"
BINARY	Byte Array	Not supported
TIMESTAMP	Timestamp with nanosec	132500 , '2012-01-0203:04:05.123456789'

Complex

Type	Description	Examples
ARRAY	Ordered collection of fields of same type	array(1,2)
MAP	Unordered collection of k/v pairs	map('a',1,'b',2)
STRUCT	Collection of named fields	struct('a', 1, 1.0). Created like STRUCT

Tables

A Hive table is logically made up of the data being stored and the associated metadata describing the layout

of the data in the table.

Hive defaults moves the data into the warehouse unless is a *external table*

```
CREATE TABLE managed_table (dummy STRING);
LOAD DATA INPATH '/user/tom/data.txt' INTO table managed_table;
```

Will move hdfs://user/tom/data.txt into the warehouse. If the table is later dropped, using:

```
DROP TABLE managed_table;
```

then the table, including its metadata **and its data is deleted**. Using *EXTERNAL* it doesn't move it nor check if it exists. If you drop it, it doesn't remove the data.

Partitions

Hive organizes tables into partitions based on the value of a *partition column* (a way to organize columns based on common values like the year in a timestamp field). Subpartitions are also common.

Partitions at table creation time using **PARTITIONED BY**

```
CREATE TABLE logs (ts BIGINT, line STRING)
PARTITIONED BY (dt STRING, country STRING);
```

Then, when loaded:

```
LOAD DATA LOCAL INPATH 'input/hive/partitions/file1'
INTO TABLE logs
PARTITION (dt='2001-01-01', country='GB');
```

At filesystem level it will create nested subdirectories. We can ask Hive for the partitions with the order `SHOW PARTITIONS;`

Buckets

More efficient queries by joining to tables that are bucketed on the same columns and make sampling more efficient. If the data is sorted if even more efficient. The following command specifies columns and number of buckets.

```
CREATE TABLE bucketed_users (id INT, name STRING)
CLUSTERED BY (id) INTO 4 BUCKETS;
```

To populate, we need to set `hive.enforce.bucketing` to **true** and then using an insert with the current table. Each bucket will be a file in the table:

```
INSERT OVERWRITE TABLE bucketed_users
SELECT * FROM users;
```

Storage formats

Hive table has two dimensions: *row format* (how rows and files are stored) and *file format* (container format fields in a row)

Default storage format: delimited text

Default row delimiter is Ctrl-A. Default collection item delimiter is Ctrl+B

Binary Storage formats: Sequence files and RCFiles

SequenceFile is default wen declare `STORED AS SEQUENCEFILE`. It supports **splittable compression**. Fields in each row are stored together.

RCFile (Record Columnar File) store in column oriented format. Permits columns that are not accessed in a query to be skipped

Importing Data

INSERT OVERWRITE TABLE

```
INSERT OVERWRITE TABLE target
SELECT col1, col2 FROM source
```

For partitioned tables you can specify the partition to insert into:

```
INSERT OVERWRITE TABLE target
PARTITION (dt='2010-01-01')
SELECT col1, col2 FROM source;
```

Multitable insert

```
FROM source
INSERT OVERWRITE TABLE target
    SELECT year, COUNT(DISTINCT station)
    GROUP BY year
INSERT OVERWRITE TABLE records_by_year
    SELECT year, COUNT(1)
    GROUP BY year
INSERT OVERWRITE TABLE good_records_by_year
    SELECT year, COUNT(1)
    WHERE temperature != 9999
    AND (quality = 0 OR quality = 1)
    GROUP BY year;
```

Altering tables

Renaming (also moves the files:

```
ALTER TABLE source RENAME TO target;
```

Dropping tables

Deletes data and metadata:

```
DROP TABLE target;
```

Deletes table contents: simply delete the files in the warehouse

Querying Data

Sorting and Aggregating

- Sorting:
  - **ORDER BY** (setting reducers to one, inefficient with very large datasets)
  - **SORT BY**: Produces a sorted file per reducer

To control which reducer a particular row goes to, **DISTRIBUTE BY** can execute subsequent aggregation:

```
hive> FROM records2
        SELECT year, temperature
        DISTRIBUTE BY year
        SORT BY year ASC, temperature DESC;

1949 111
1949 78
1950 22
1950 0
1950 -11
```

MapReduce Scripts

Using Hadoop Streaming, the **TRANSFORM**, **MAP** and **REDUCE** clauses make it possible to invoke an external script

hive> ADD FILE /path/to/is_good_quality.py; FROM records2 SELECT TRANSFORM(year, temperature, quality) USING 'is_good_quality.py' AS year, temperature;

Joins

Inner joins

Each match in the input tables results in a row in the output. Some DDBB allows to performs the JOIN in the WHERE clause. However is not possible in Hive at it must be write explicitly.

```
SELECT sales.*, things.*
    FROM sales JOIN things ON (sales.id = things.id);
```

You can see how any MR jobs Hive will use for a query prefixing with the **EXPLAIN** clause (or **EXPLAIN EXTENDED**):

```
EXPLAIN
SELECT sales.*, things.*
FROM sales JOIN things ON (sales.id = things.id);
```

Outer joins

Allow to return also nonmatches of the *left* or *right* table. **LEFT OUTER JOIN**, **RIGHT OUTER JOIN**:

```
SELECT sales.*, things.*
    FROM sales LEFT OUTER JOIN things ON (sales.id = things.id)
```

Semi joins

```
SELECT *
FROM things
LEFT SEMI JOIN sales
ON (sales.id = things.id);
```

Map joins

If one table is small enough to fit in memory, Hive can load it to perform the join in each of the mappers.

```
SELECT /*+ MAPJOIN(things) */ sales.*, things.*
FROM sales JOIN things ON (sales.id = things.id)
```

Subqueries

Hive only permits a subquery in the FROM clause of a SELECT:

```
SELECT *
FROM (
    SELECT *
    FROM records
    WHERE temp = 20) mt
GROUP BY station, year;
```

Views

```
CREATE VIEW valid_records
AS
SELECT *
FROM records2
WHERE temp = 20
```

When creating a view, the Hive query is not run but stored in the metastore. Views are included in the **SHOW TABLES** and the **DESCRIBE EXTEND view** gives even the view query.

User defined functions

Written in Java or using **SELECT TRANSFORM** if others used. Three types

- UDF's (regular): Operates in one row and produce one row too
- UDAFs (user-defined aggregate functions): Multiple input rows and a single row output
- UDTFs (user-defined table-generating functions): Operates in one row and produces multiple rows (a table)

Writing an UDF

A class must extends org.apache.hadoop.hive.ql.exec.UDF and implement at least one **evaluate()** method. Then package as JAR, register it in Hive and create an alias:

```
ADD JAR /path/to/hive-examples.jar
CREATE TEMPORARY FUNCTION strip AS 'com.example.hive'
```

TEMPORARY means that the function is only defined during this session

Writing an UDAF

The implementation has to be capable of combining partial aggregations into a final result. It must subclass **org.apache.hive.ql.exec.UDAF** and contain one or more nested static classes implementing **org.apache.hadoop.hive.ql.exec.UDAFEvaluator**. An evaluator must implements:

- **init()** resets internal state
- **iterate()** called every time a new value to be aggregated
- **terminatePartial()** called when Hive wants a result for the partial aggregation
- **merge()** called when Hive decides to combine aggregations
- **terminate()** called when the final result of the aggregation is needed

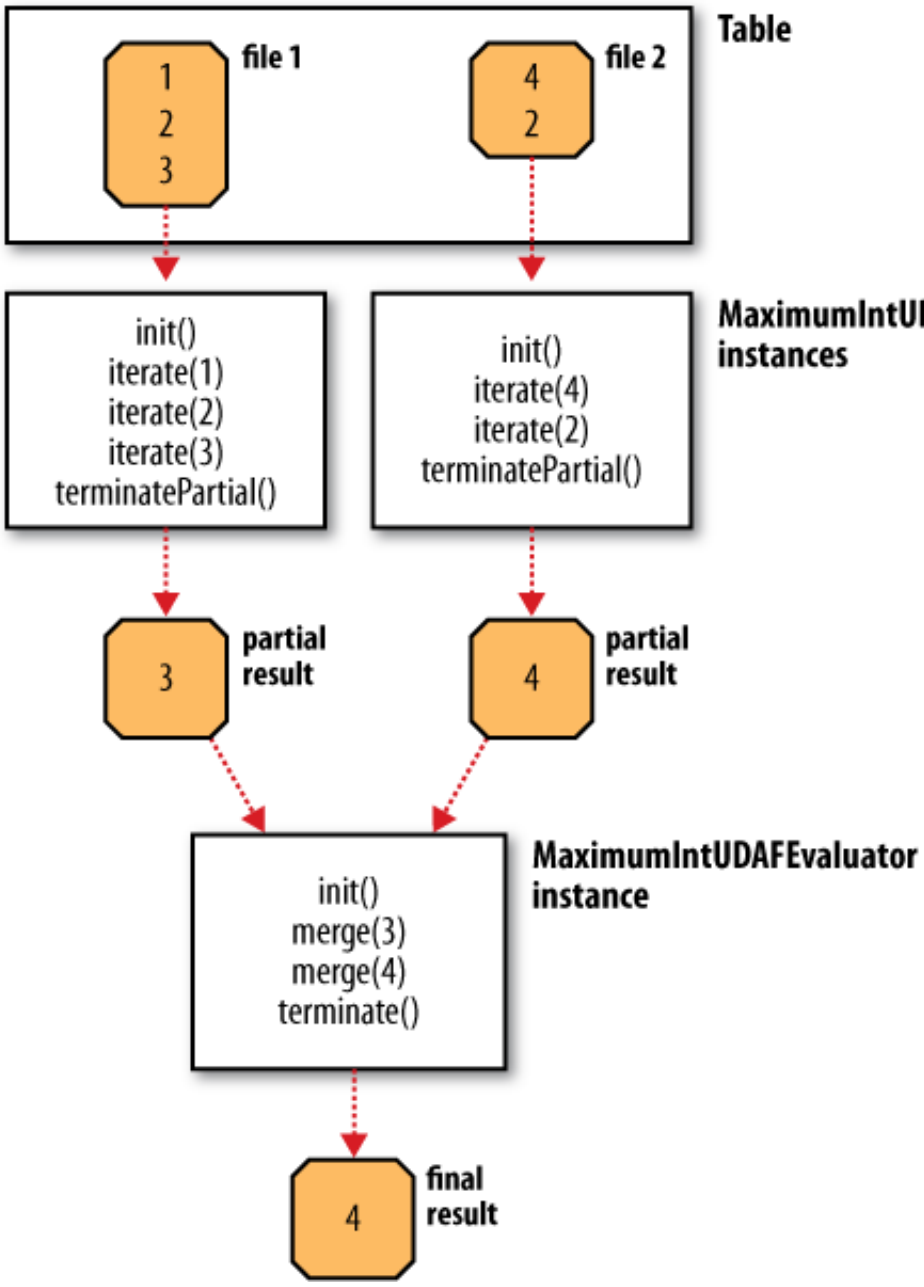


Table
file 1
1
2
3
file 2
4
2
init()
iterate(1)
iterate(2)
iterate(3)
terminatePartial()
init()
iterate(4)
iterate(2)
terminatePartial()
MaximumIntUDAFEvaluator
instances
3
partial
result
4
partial
result
init()
merge(3)
merge(4)
terminate()
MaximumIntUDAFEvaluator
instance
4
final
result

HBase

HBasics

Distributed column oriented database built on top of HDFS used when real-time read/write random-access is required at large scale. Doesn't support SQL but scales linearly (adding a new node).

Concepts

Cells are timestamped by HBase at insertion. Row key are bytes arrays so anything can serve as row key (strings or binary representations).

Row columns are grouped into *column families* with a common prefix. Tables are partitioned horizontally into *regions* of configured size when it splits into equal size. Until this split happens, all loading will be against a single server.

Row updates are atomic.

Implementation

A *master* node manages many *regionservers* slaves and uses a Zookeeper server by itself.

Regionservers slaves nodes are in *conf/regionservers* much like the *conf/slaves* of Hadoop.

Persists data throught Hadoop API so it can write to KFS, Amazon's S3 and HDFS

HBase in operation

- **-ROOT-**: Holds the list of .META. table regions. Is the first thing clients learn when connected to ZooKeeper
- **.META.**: Holds the list of all user-space regions

Test Drive

```
$ start-hbase.sh
$ hbase shell
```

Will launch a loca HBase on **/tmp/hbase-${USERID}**

To **create a table** use:

```
hbase> create 'test', 'data'
```

To **list** tables use:

```
hbase> list
```

To **insert data** use:

```
hbase> put 'test', 'row1', 'data:1', 'value1'
```

To **remove the table** use:

```
hbase> disable 'test'
hbase> drop 'test'
```

Clients

Java

First gets a org.apache.hadoop.conf.Configuration instance with HBase configuration read from *hbase-site.xml* and and *hbase-default.xml* to creates **HBaseAdmin** for administering and adding and dropping tables **HTable** for access tables.

Avro, REST and Thr.ift

Are slower as a java server is parsing request to HBase.

- **REST**: **Stargate** is initialized with `hbase-daemon.sh start rest` on 8080
- **Thrift**: Is started with `hbase-daemon.sh start thrift` on 9090

- **Avro**: Is started with `hbase-daemon.sh start avro` on 9090 too.

Core features

- **No real indexes**: as row and columns are stored sequentially. Insert performance is based on table size.
- **Automatic partitioning**: As the tables grow they'll split automatically and be distributed across available nodes.
- **Scale linearly and automatically with new nodes**: Add a node, point it to the Zookeeper and regions will rebalance automatically.
- **Commodity hardware**: Nodes of 1000-5000$ nodes rather than 50000$ ones.
- **Fault tolerance**: No worries about individual node downtime
- **Batch processing**: Allow fully parallel distributed jobs

Common problems

Versions

HBase version compatibility with Hadoop is matched up to 0.90 so HBase 0.20.5 would run on an Hadoop 0.20.2 but HBase 0.19.5 would not. 0.90 will work on Hadoop 0.20.x and 0.21.x

HDFS

HBase uses HDFS in a different way that Hadoop.

- **Running out of file descriptors** is when, for example, you have 2000 files opened (100 regions x 10 columns x 2 flush files + other files) and the default limit of file descriptors per process ir 1024
- **Running out of datanode threads**: Hadoop datanode can't run more than 256 threads unless you set a higher limit in `dfs.datanode.max.xcievers`
- **Sync**: You must run HBase on an HDFS that has a working sync. Otherwise, you will lose data. This means running HBase on Hadoop 0.20.205.0 or later

UI and metrics

60010 is the default port and displays a list of basic attributes. Enabling Hadoop metrics and tying them to

Ganglia will give views of what's happening on the cluster

Schema design

HBase has versioned cells, sorted rows and capacity to add columns on the fly. These factors should be considered when designing schemas but specially how the data will be accessed. Also with column-oriented stores can host very populated tables at no incurred cost.

Joins

No native join in HBase but with wide tables there is no need

Row keys

Take time designing the row key

Counters

With **incrementColumnValue()** a counter can the incremented many thousands of times a second

Sqoop

A sample import

Databases supported include:

- MySQL
- PostgreSQL
- Oracle
- SQL Server
- DB2

Example import:

```
$ sqoop import --connect jdbc:mysql://localhost/db --table widgets -m 1
```

The `-m 1` will use only 1 map task to get a single file in HDFS. It will import in text format, however, it is possible to use a binary format.

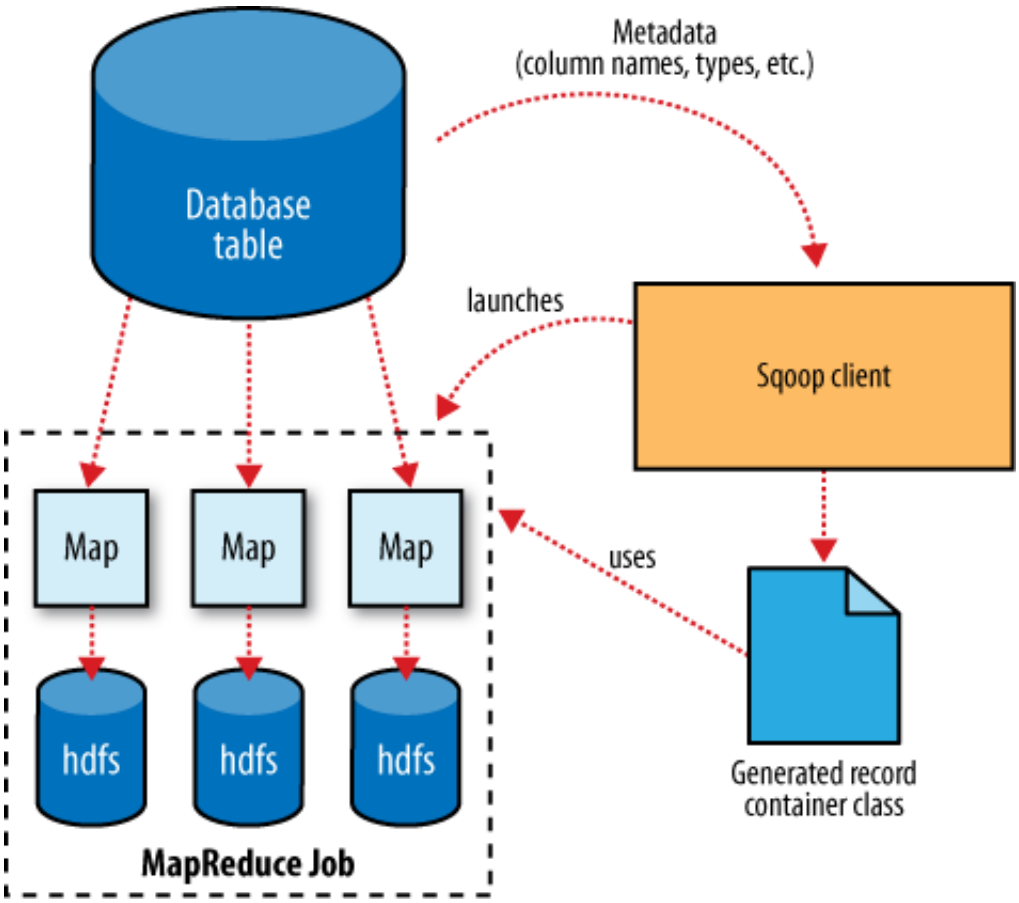
Generated code

When you do the import, it generates a *widget.java* written to the current local directory. This class is able of holding a record from DDBB, manipulate it in MR or store it in a SequenceFile in HDFS.

You can also generate the source code without importing importing the table with the command `sqoop codegen`.

Sqoop also supports Avro

Database Imports: A Deeper Look



Sqoop examines the table, get all columns and types to map the to Java types and create a model of the table. Then, with the **readFields(ResultSet res)** and the **write(PreparedStatement s)** reads and writes the ddbb

Hive tables throught Sqoop

```
$ sqoop create-hive-table --connect jdbc:mysql://localhost/my_db --table widgets --fiels-terminate

% hive> LOAD DATA INPATH "widgets" INTO TABLE widgets;
```

In short:

1. Import data into HDFS using Sqoop
2. Create a Hive table
3. Load the data into Hive

It can be done in a unique pass:

```
$ sqoop import jdbc:mysql://localhost/my_db --table widgets -m 1 --hive_import
```

Importing Large Objects (CLOB, BLOB)

Sqoop import large objects into a LobFile to access a field without accessing the record (it can be truly huge)

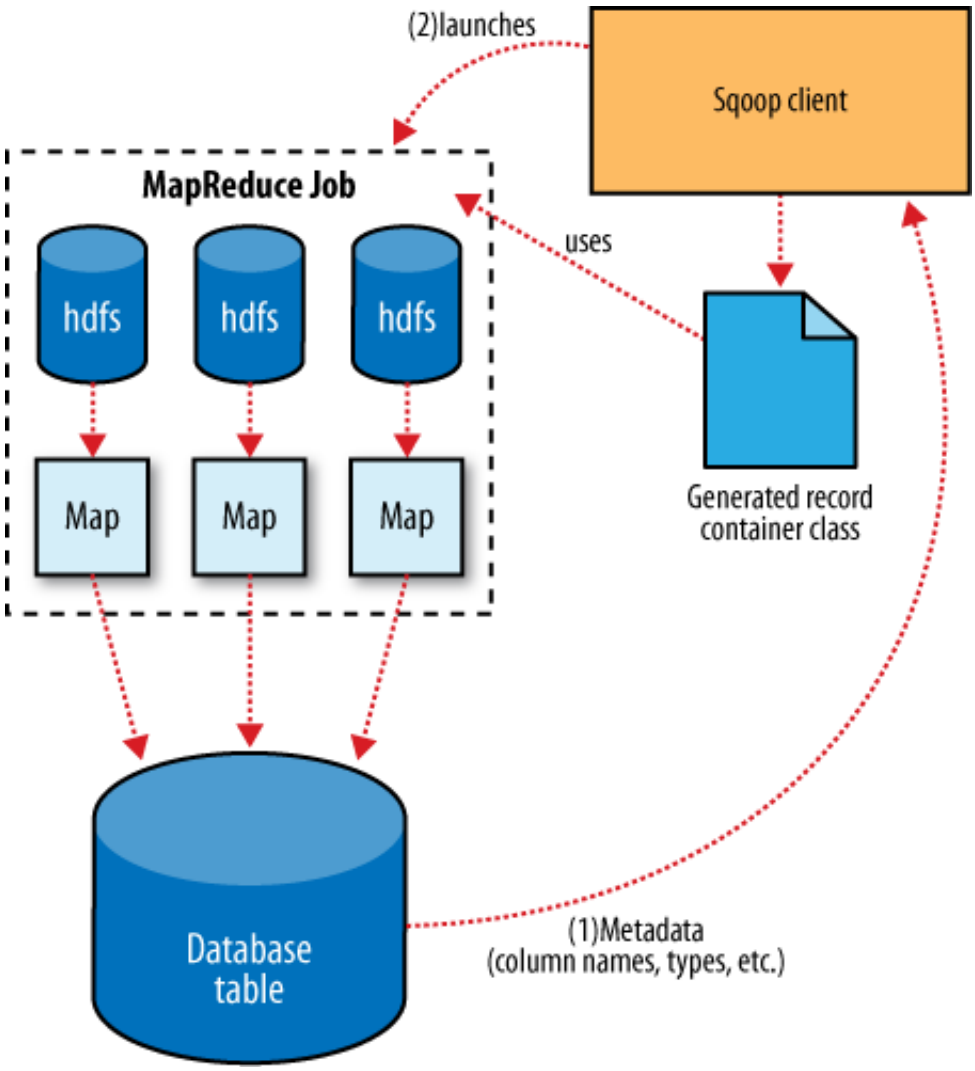
Performing an export

The database must be prepared and the data types must be set explicitly when creating the MySQL table. The command:

```
$ sqoop export --connect jdbc:mysql://localhost/my_db -m 1 --table sales --export-dir /user/hive/w
```

The `--input-fields-terminated '\0001'` is the default delimiter that is used in Hive

Exports: A Deeper Look



The export process is similar: some parallel tasks performs queries to the MySQL ddbb but they are nor ordered or atomic operations. Sqoop can also exports records stored in SequenceFiles

ZooKeeper

Features

- **Simple**: ZooKeeper at its core, a filesystem that exposes a few simple operations and some abstractions such as ordering and notifications.
- **Expressive**: ZooKeeper primitives are building blocks that can build a coordination data structure and protocol.
- **Highly available**: Is run on a collection of machines to be highly available.
- **Facilitates loosely coupled interactions**: Interactors that doesn't need to know about each other
- **Is a library**: If common coordination patterns.

Installing and running

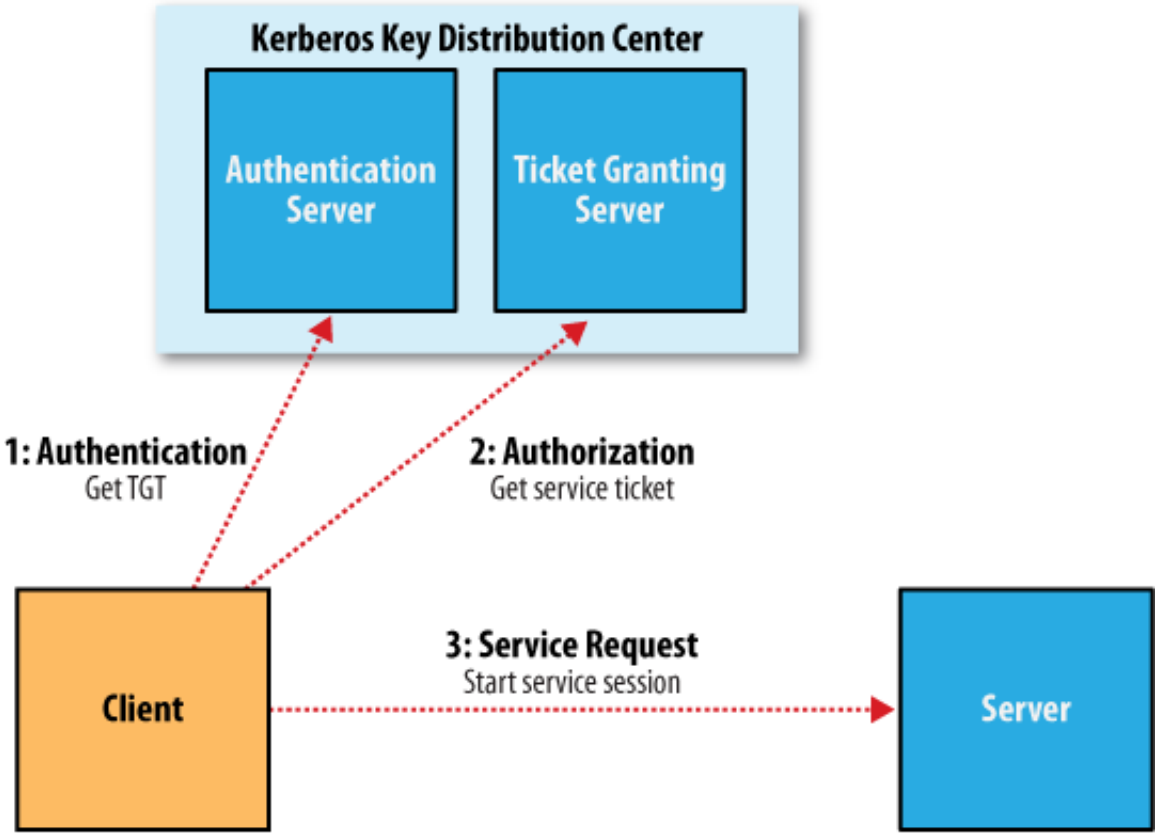
We need to set up a conf file usually called *zoo.conf*:

```
tickTime=2000
dataDir=/Users/mac/zookeeper
clientPort=2181
```

Ticktime is the time in miliseconds for heartbeats, dataDir is the persisten data directory for Zookeeper and the clientPort is the connection port for Zookeeper clients. With this, we can start the server:

```
$ zkServer.sh start
```

Several commands are available to send to the server using **nc**:



The ZooKeeper Service

Data Model

Zookeeper maintains a hierarchical tree of nodes called znodes to store data. Zookeeper is designed for coordination of small size files.

Data access is atomic. The references are by paths like filesystems in Unix, so they aren't URI's.

Ephemeral and persistent znodes

An Ephemeral is deleted when the creating client's session ends. A persisten znode is not tied to any client.

Sequence numbers

A *sequential* znode has a number on its name and can be incremented. So if we create a node calle */a/b-* it may be called */a/b-1* and the next will be called */a/b-2*

Watches

Allows to clients to get notifications when a znode changes in some way

Operations

Operation	Description
create	Creates a znode (parent must already exist)
delete	Deletes a znode
exists	Tests wether a znode exists and retrieves its metadata
getACL, setACL	Get/sets the ACL for a node
getChildren	Gets a list of the children of a znode
getData, setData	Gets/sets the data associated with a znode
sync	Synchronizes a client's view of a znode with Zookeeper

Update operations are conditional. **delete** or **setData** has to specify the version number of the znode that is being updated (found from **exists** call)

Multi-update

multi is used to batch multiple primitive operations

Watch triggers

exists, getChildren and **getData** may have watches (describe earlier) set on them.

Table 14-3. Watch creation operations and their corresponding triggers

Watch creation	Watch trigger				
	create		delete		setData
	znode	child	znode	child	
exists	NodeCreated		NodeDeleted		NodeDataChanged
getData			NodeDeleted		NodeDataChanged
getChildren		NodeChildrenChanged	NodeDeleted	NodeChildrenChanged	

ACLs

A znode is created with a list of ACLs, which determines who can perform certain operations on it. It

depends on authentication:

1. *Digest*: Client is authenticated by a user and pass
2. *Sasl*: Client is authenticated using Kerberos
3. *Ip*: Client is authenticated by its IP address

Implementation

With replication, it can provides service as long as the majority of the ensemble are up (3 of 5, 2 of 3).

Its work is simple, it ensure that the every modification to the tree znodes is replicated to a majority of the ensemble using a protocol called Zab that runs in two phases:

1. *Leader election*
2. *Atomic broadcast*: All write requests are forwarded to the leader which broadcasts the update to the followers.

Consistency

Every update made to the znode tree is given a globally unique id called **zxid**.

- *Sequential consistency*: Updates from any particular client are applied in the order that they are sent.
- *Atomicity*: Updates either succeed or fail.
- *Single system image*: A client will see the same view of the system regardless of the server it connects to
- *Durability*: Once an update has succeeded, it will persist and will not be undone
- *Timeliness*: The lag in any client's view of the system is bounded so it will not be out of date by more than some multiple of tens of seconds.

Sessions

A client tries to connect to the first Zookeeper of the list, if fails will go to the next until none is available.

Once connected a new session is created with a timeout period that a client must avoid by sending ping requests.

Time

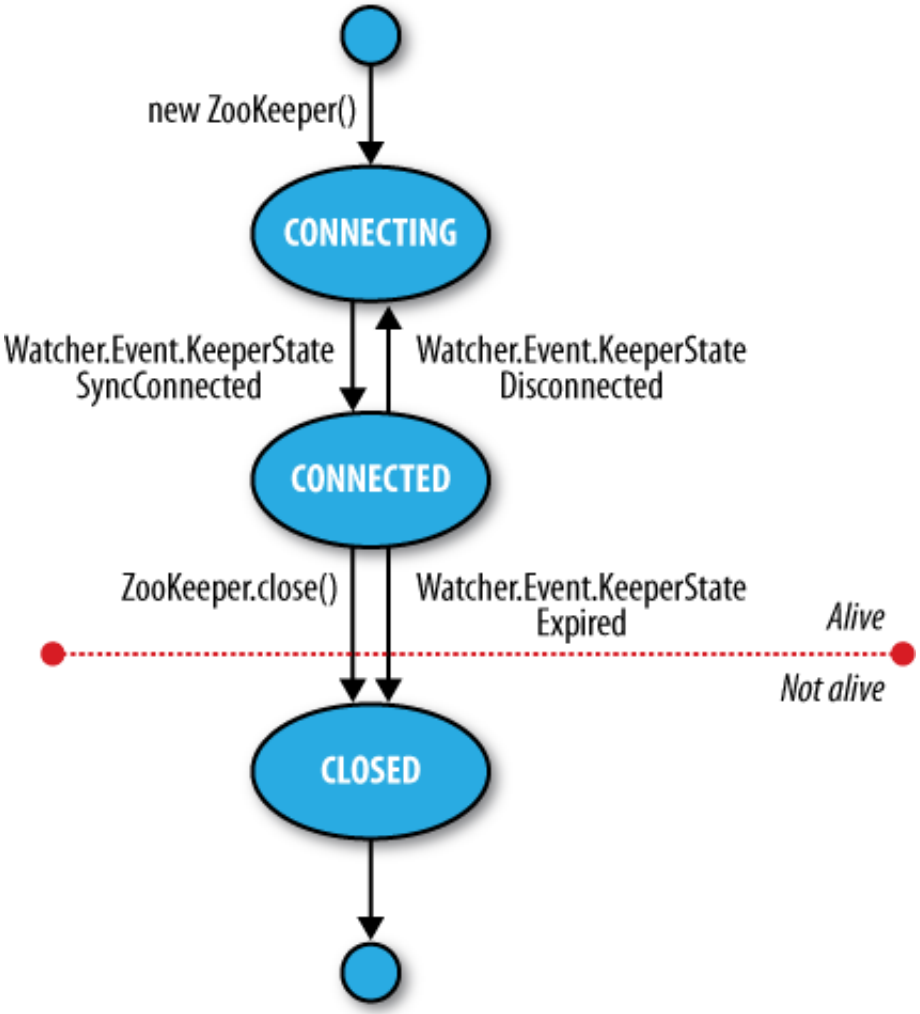
A *thick time* is usually of 2 seconds that translates to an allowable session timeout between 4 and 40

seconds.

States

Zookeeper object transitions throught different states in its lifecycle. You can query its state with **getState()**

States is an enum representing the different states.



A client using the **Zookeeper** object can receive notifications via a **Watcher** objects.

Building Application with ZooKeeper

A Configuration Service

Common pieces of configuration information can be shared by machines.

The Resilient ZooKeeper Application

The programs so far have been assuming a reliable network so when they run on a real network, they can fail in several ways:

- **InterruptedException** thrown when an operation is interrupted but not neccesary is that it failed. Is a call to **interrupt()** method on the thread
- **KeeperException**: Is thrown if the ZooKeeper server signals an error or if there is a communication problem with the server.
- **State exceptions**: When the operations fails because it cannot be applied to the znode tree.
- **Recoverable Exceptions**: The app can recover from this exception. Is thrown by **KeeperException.ConnectionLossException** which means that the connection to ZooKeeper has been lost.
- **Unrecoverable exceptions**:

A Lock Service

Distributed lock is a mechanism for providing mutual exclusion between a collection of processes.

More Distributed Data Structures and Protocols

BookKeeper and Hedwig

A highly available an reliable logging service. It can be used to provive write-ahead logging (the log is written before the operation).

BookKeeper clients create logs called ledgers and each record is called a *ledger entry* (a byte array). Hadoop namenode writes its edit log to multiple disks, one of which is tipically an NFS.

Hedwig is a topic-based publish-subscribe system build on BookKeeper

ZooKeeper in Production

Resilience and Performance

ZooKeeper machines should be located to minimize the impact of machine and network failure. This means

that servers should be spread across racks, power supplies, and switches, so that the failure of any one of these does not cause the ensemble to lose a majority of its servers.

ZooKeeper has an observer nodes, which is like a non-voting follower. They allow a ZooKeeper cluster to improve read performance without hurting write performance. They allow a ZooKeeper cluster to span data centers without impacting latency as much as regular voting followers by placing the voting members in one data center and observers in the other.

Configuration

Each server has a numeric identifier that is unique and must fall between 1 and 255. Is specified in plain text in a file named myid in the directory specified by the **dataDir** property.

We also need to give all the servers all the identities and network locations of the others in the ensemble. The ZooKeeper configuration file must include a line for each server, of the form: server.n=hostname:port:port

The value of n is replaced by the server number. The first port is the port that followers use to connect to the leader, and the second is used for leader election. Example: tickTime=2000 dataDir=/disk1/zookeeper dataLogDir=/disk2/zookeeper clientPort=2181 initLimit=5 syncLimit=2 server.1=zookeeper1:2888:3888 server.2=zookeeper2:2888:3888 server.3=zookeeper3:2888:3888

Servers listen on three ports: 2181 for client connections; 2888 for follower connections, if they are the leader; and 3888 for other server connections during the leader election phase

Hortonworks We Do Hadoop

Cheat Sheet
Hive for SQL Users

Contents

If you're already a SQL user then working with Hadoop may be a little easier than you think, thanks to Apache Hive. Apache Hive is data warehouse infrastructure built on top of Apache™ Hadoop® for providing data summarization, ad hoc query, and analysis of large datasets. It provides a mechanism to project structure onto the data in Hadoop and to query that data using a SQL-like language called HiveQL (HQL).

Use this handy cheat sheet (based on this original MySQL cheat sheet) to get going with Hive and Hadoop.

Additional Resources

Learn to become fluent in Apache Hive with the Hive Language Manual:
https://cwiki.apache.org/confluence/display/Hive/LanguageManual

Hortonworks Sandbox

Get in the Hortonworks Sandbox and try out Hadoop with interactive tutorials:
http://hortonworks.com/sandbox

Hortonworks University

Register today for Apache Hadoop Training and Certification at Hortonworks University:
http://hortonworks.com/training

3460 West Bayshore Rd.
Palo Alto, CA 94303 USA
US: 1.855.846.7866
International: 1.408.916.4121
www.hortonworks.com

Twitter: twitter.com/hortonworks
Facebook: facebook.com/hortonworks
LinkedIn: linkedin.com/company/hortonworks

Hortonworks

Hortonworks We Do Hadoop

Query

Function	MySQL	HiveQL
Retrieving information	`SELECT from_columns FROM table WHERE conditions;`	`SELECT from_columns FROM table WHERE conditions;`
All values	`SELECT * FROM table;`	`SELECT * FROM table;`
Some values	`SELECT * FROM table WHERE rec_name = “value”;`	`SELECT * FROM table WHERE rec_name = "value";`
Multiple criteria	`SELECT * FROM table WHERE rec1=”value1” AND rec2=”value2”;`	`SELECT * FROM TABLE WHERE rec1 = "value1" AND rec2 = "value2";`
Selecting specific columns	`SELECT column_name FROM table;`	`SELECT column_name FROM table;`
Retrieving unique output records	`SELECT DISTINCT column_name FROM table;`	`SELECT DISTINCT column_name FROM table;`
Sorting	`SELECT col1, col2 FROM table ORDER BY col2;`	`SELECT col1, col2 FROM table ORDER BY col2;`
Sorting backward	`SELECT col1, col2 FROM table ORDER BY col2 DESC;`	`SELECT col1, col2 FROM table ORDER BY col2 DESC;`
Counting rows	`SELECT COUNT(*) FROM table;`	`SELECT COUNT(*) FROM table;`
Grouping with counting	`SELECT owner, COUNT(*) FROM table GROUP BY owner;`	`SELECT owner, COUNT(*) FROM table GROUP BY owner;`
Maximum value	`SELECT MAX(col_name) AS label FROM table;`	`SELECT MAX(col_name) AS label FROM table;`
Selecting from multiple tables (Join same table using alias w/”AS”)	`SELECT pet.name, comment FROM pet, event WHERE pet.name = event.name;`	`SELECT pet.name, comment FROM pet JOIN event ON (pet.name = event.name);`

Metadata

Function	MySQL	HiveQL
Selecting a database	`USE database;`	`USE database;`
Listing databases	`SHOW DATABASES;`	`SHOW DATABASES;`
Listing tables in a database	`SHOW TABLES;`	`SHOW TABLES;`
Describing the format of a table	`DESCRIBE table;`	`DESCRIBE (FORMATTED|EXTENDED) table;`
Creating a database	`CREATE DATABASE db_name;`	`CREATE DATABASE db_name;`
Dropping a database	`DROP DATABASE db_name;`	`DROP DATABASE db_name (CASCADE);`

3460 West Bayshore Rd.
Palo Alto, CA 94303 USA
US: 1.855.846.7866
International: 1.408.916.4121
www.hortonworks.com

Twitter: twitter.com/hortonworks
Facebook: facebook.com/hortonworks
LinkedIn: linkedin.com/company/hortonworks

Hortonworks

Hortonworks We Do Hadoop

Current SQL Compatibility

Hive SQL Datatypes	Hive SQL Semantics
INT	SELECT, LOAD INSERT from query
TINYINT/SMALLINT/BIGINT	Expressions in WHERE and HAVING
BOOLEAN	GROUP BY, ORDER BY, SORT BY
FLOAT	Sub-queries in FROM clause
DOUBLE	GROUP BY, ORDER BY
STRING	CLUSTER BY, DISTRIBUTE BY
TIMESTAMP	ROLLUP and CUBE
BINARY	UNION
ARRAY, MAP, STRUCT, UNION	LEFT, RIGHT and FULL INNER/OUTER JOIN
DECIMAL	CROSS JOIN, LEFT SEMI JOIN
CHAR	Windowing functions (OVER, RANK, etc)
CARCHAR	INTERSECT, EXCEPT, UNION, DISTINCT
DATE	Sub-queries in WHERE (IN, NOT IN, EXISTS/ NOT EXISTS)
	Sub-queries in HAVING

Color Key
Hive 0.10
Hive 0.11
FUTURE

Command Line

Function	Hive
Run query	hive -e 'select a.col from tab1 a'
Run query silent mode	hive -S -e 'select a.col from tab1 a'
Set hive config variables	hive -e 'select a.col from tab1 a' -hiveconf hive.root.logger=DEBUG,console
Use initialization script	hive -i initialize.sql
Run non-interactive script	hive -f script.sql

Hive Shell

Function	Hive
Run script inside shell	source file_name
Run ls (dfs) commands	dfs -ls /user
Run ls (bash command) from shell	!ls
Set configuration variables	set mapred.reduce.tasks=32
TAB auto completion	set hive.<TAB>
Show all variables starting with hive	set
Revert all variables	reset
Add jar to distributed cache	add jar jar_path
Show all jars in distributed cache	list jars
Delete jar from distributed cache	delete jar jar_name

3460 West Bayshore Rd.
Palo Alto, CA 94303 USA
US: 1.855.846.7866
International: 1.408.916.4121
www.hortonworks.com

Twitter: twitter.com/hortonworks
Facebook: facebook.com/hortonworks
LinkedIn: linkedin.com/company/hortonworks

Hortonworks

CCD-410 Study Guide

This exam focuses on engineering data solutions in MapReduce and understanding the Hadoop ecosystem (including Hive, Pig, Sqoop, Oozie, Crunch, and Flume). Candidates who successfully pass CCD–410 are awarded the Cloudera Certified Hadoop Developer (CCDH) credential.

Recommended Cloudera Training Course

Cloudera Developer Training for Apache Hadoop

Practice Test

CCD–410 Practice Test Subscription

Exam Sections

Each candidate receives 50 - 55 live questions. Questions are delivered dynamically and based on difficulty ratings so that each candidate receives an exam at a consistent level. Each test also includes at least five unscored, experimental (beta) questions.

Infrastructure: Hadoop components that are outside the concerns of a particular MapReduce job that a developer needs to master (25%)
Data Management: Developing, implementing, and executing commands to properly manage the full data lifecycle of a Hadoop job (30%)
Job Mechanics: The processes and commands for job control and execution with an emphasis on the process rather than the data (25%)
Querying: Extracting information from data (20%)

1. Infrastructure Objectives

- Recognize and identify Apache Hadoop daemons and how they function both in data storage and processing.
- Understand how Apache Hadoop exploits data locality.
- Identify the role and use of both MapReduce v1 (MRv1) and MapReduce v2 (MRv2 / YARN) daemons.
- Analyze the benefits and challenges of the HDFS architecture.
- Analyze how HDFS implements file sizes, block sizes, and block abstraction.
- Understand default replication values and storage requirements for replication.
- Determine how HDFS stores, reads, and writes files.
- Identify the role of Apache Hadoop Classes, Interfaces, and Methods.
- Understand how Hadoop Streaming might apply to a job workflow.

2. Data Management Objectives

- Import a database table into Hive using Sqoop.
- Create a table using Hive (during Sqoop import).
- Successfully use key and value types to write functional MapReduce jobs.
- Given a MapReduce job, determine the lifecycle of a Mapper and the lifecycle of a Reducer.
- Analyze and determine the relationship of input keys to output keys in terms of both type and number, the sorting of keys, and the sorting of values.
- Given sample input data, identify the number, type, and value of emitted keys and values from the Mappers as well as the emitted data from each Reducer and the number and contents of the output file(s).
- Understand implementation and limitations and strategies for joining datasets in MapReduce.
- Understand how partitioners and combiners function, and recognize appropriate use cases for each.
- Recognize the processes and role of the the sort and shuffle process.
- Understand common key and value types in the MapReduce framework and the interfaces they implement.
- Use key and value types to write functional MapReduce jobs.

3. Job Mechanics Objectives

- Construct proper job configuration parameters and the commands used in job submission.
- Analyze a MapReduce job and determine how input and output data paths are handled.
- Given a sample job, analyze and determine the correct InputFormat and OutputFormat to select based on job requirements.
- Analyze the order of operations in a MapReduce job.
- Understand the role of the RecordReader, and of sequence files and compression.
- Use the distributed cache to distribute data to MapReduce job tasks.
- Build and orchestrate a workflow with Oozie.

4. Querying Objectives

- Write a MapReduce job to implement a HiveQL statement.
- Write a MapReduce job to query data stored in HDFS.