Outlines

Overviews and Problems of Hadoop + HDFS

In this section, we give a brief view of MapReduce, Hadoop and then show some shortcomings of Hadoop+HDFS[1]. Firstly, we give a general overview about MapReduce, explain what it is and achieves and how it works; Secondly, we introduce Hadoop as a implementation of MapReduce, show general overview of Hadoop and Hadoop Distributed File System (HDFS), explain how it works and what tasks-assigned strategy is; Thirdly, we list a few major flaws and drawbacks when using Hadoop+HDFS as a platform. These flaws are the reasons we do this research and study.

MapReduce and Hadoop overviews

MapReduce[2] is a programming model (Figure 1) and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. MapReduce provides automatic parallelization and distribution, fault-tolerance, I/O scheduling, status and monitoring.

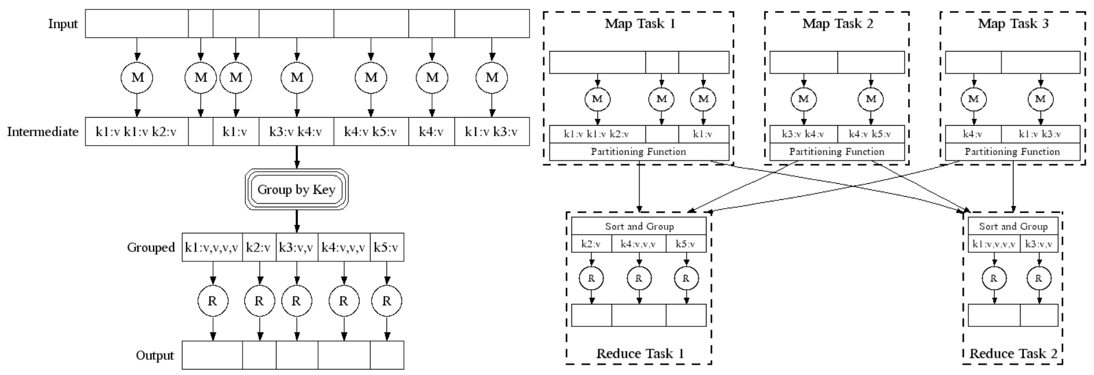


Figure 1: MapReduce Parallel Execution

Hadoop[3] implements MapReduce, using HDFS for storage. MapReduce divides applications into many small blocks of work. HDFS creates multiple replicas of data blocks for reliability, placing them on compute nodes around the cluster. MapReduce can then process the data where it is located.

Hadoop execution engine is divided into three main modules (Figure 2): JobTracker[4], TaskTracker[5] and Applications.

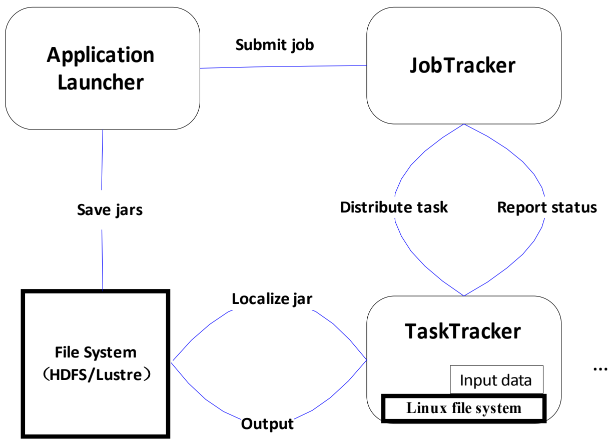


Figure 2: Hadoop Execution Engine

As the graph shows, Application Launcher submits job to JobTracker and saves jars of the job in file system. JobTracker divide the job into several map and reduce tasks. Then JobTracker schedule tasks to TaskTracker for execution. TaskTracker fetched task jar from the file system and save the data into the local file system. Hadoop system is based on centralization schedule mechanism. JobTracker is the information center, in charge of all jobs' execution status and make schedule decision. TaskTracker will be run on every node and in charge of managing the status of all tasks which running on that node. And each node may run many tasks of many jobs.

Applications typically implement the Mapper[6] and Reducer[7] interfaces to provide the map and reduce methods. Mapper maps input key/value pairs to a set of intermediate key/value pairs, Reducer reduces a set of intermediate values which share a key to a smaller set of values. And, Reducer has 3 primary phases: *shuffle*, *sort* and *reduce*.

Shuffle: Input to the Reducer is the sorted output of the mappers. In this phase the framework fetches the relevant partition of the output of all the mappers, via HTTP.

Sort: The framework groups Reducer inputs by keys (since different mappers may have output the same key) in this stage.

Reduce: In this phase the reduce() method is called for each <key, (list of values)> pair in the grouped inputs.

According to Hadoop tasks-assigned strategy, it is TaskTracker's duty to ask tasks to execute, rather than JobTacker assign tasks to TaskTrackers. Actually, Hadoop will maintain a topology map of all nodes in cluster organized as Figure 3. When a job was submitted to the system, a thread will pre-assign task to node in the topology map. After that, when TaskTracker send a task request heartbeat to JobTracker, JobTracker will select a fittest task according to the former pre-assign.

As a platform, Hadoop provides services, the ideal situation is that it uses resources (CPU, memory, etc) as little as possible, left resources for applications. We sum up two kinds of typical applications or tests which must deal with huge inputs:

- General statistic applications (such as: *WordCount*[8])
- Computational complexity applications (such as: *BigMapOutput*[9], *webpages analytics processing*)

For the first one, general statistics applications, whose MapTask outputs (intermediate results) are small, Hadoop works well. However, for the second one, high computational complexity applications, which generate large even increasing outputs, Hadoop shows lots of bottlenecks and problems. First, write/read big MapTask outputs in local Linux file system will get OS or disk I/O bottleneck; Second, in shuffle stage, Reducer node need to use HTTP to fetch the relevant partition of the big outputs of all the MapTask before reduce phase begins, this will generate lots of net I/O and merge/spill[10] operations and also take up mass resources. Even worse, the bursting shuffle stage will make memory exhausted and make kernel kill some key threads (such as SSH) according to java's wasteful memory usage, this makes cluster unbalance and instable. Third, HDFS is time consuming for storing small files.

Challenges of Hadoop + HDFS

1. According to tasks-assigned strategy, Hadoop cannot make task is data local absolutely. For example when node1 (rack 2) requests a task, but all tasks pre-assign to this node has finished, then JobTracker will give node1 a task pre-assigned to other nodes in rack 2. In this situation, node1 will run a few tasks whose data is not locally. This break the Hadoop principle: "***Moving Computation is Cheaper than Moving Data"***. This generates huge net I/O when these kinds of tasks have huge inputs.

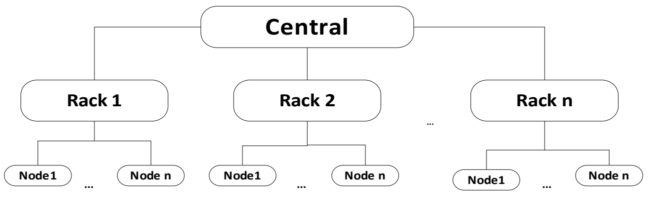


Figure 3: Topology Map of All Cluster Nodes

2. Hadoop+HDFS storage strategy of temp/intermediate data is not good for high computational complexity applications (such as web analytics processing), which generate big and increasing MapTask outputs. Save big MapTask outputs in local Linux file system will get OS/disk I/O bottleneck. These I/O should be *disperse* and *parallel* but Hadoop doesn't;
3. Reduce node need to use HTTP to shuffle all related big MapTask outputs before real task begins, which will generate lots of net I/O and merge/spill[2] operation and also take up mass resources. Even worse, the bursting shuffle stage will make memory exhausted and make kernel kill some key threads (such as SSH) according to java's wasteful memory usage, this will make the cluster unbalance and instable.
4. HDFS can not be used as a normal file system, which makes it difficult to extend.
5. HDFS is time consuming for little files.

Some useful suggestions

- ✓ Distribute MapTask intermediate results to multiple local disks. This can lighten disk I/O obstruction;
- ✓ Disperse MapTask intermediate results to multiple nodes (on Lustre). This can lessen stress of the MapTask nodes;
- ✓ Delay actual net I/O transmission time, rather than transmitting all intermediate results at the

beginning of ReduceTask (shuffle phase). This can disperse the network usage in time scale.

Hadoop over Lustre

Using Lustre instead of HDFS as the backend file system for Hadoop is one of the solutions of the above challenges. First of all, Each Hadoop MapTask can read parallelly when inputs are not locally; In the second place, saving big intermediate outputs in Lustre, which can be *dispersed* and *parallel,* will lessen mapper nodes' OS and disk I/O traffic; In the third place, Lustre create a hardlink in shuffle stage for the reducer node, which is delay actual network transmission time and more effective. Finally, Lustre is more extended which can be mounted as a normal POSIX file system and more efficient for read/write little files.

File Systems for Hadoop

In this section, we firstly talk about when and how Hadoop use file system, then we sum up three kinds of typical Hadoop I/O, and explain the generated scenes of them. After that we bring up the differences of each stage between Lustre and HDFS, and give a general difference summary table.

When Hadoop use file system

Hadoop uses two kinds of file systems： HDFS, Local Linux file system. Next, we make a list of use context of above two file system.

- ***HDFS:***
  - ✓ Write/Read job's inputs;
  - ✓ Write/Read job's outputs;
  - ✓ Write/Read job's configurations and necessary jars/dlls.
- ***Local Linux file system:***
  - ✓ Write/Read MapTask outputs (intermediate results);
  - ✓ Write/Read ReduceTask inputs (reducer fetch mapper intermediate results);
  - ✓ Write/Read all temporary files (produced by merge/spill operations).

Three Kinds Typical I/O of Hadoop+HDFS

As the figure 4 shows, there are three kinds of typical I/O: *"HDFS bytes read", "HDFS bytes write", "local bytes write/read".*

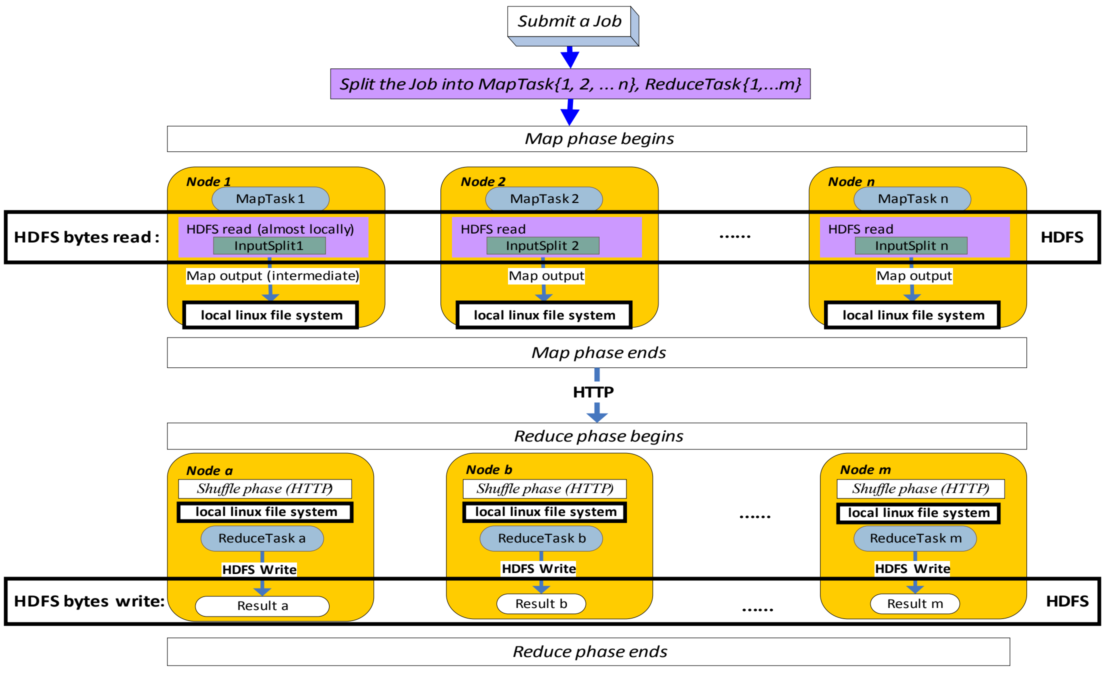


Figure 4: Hadoop+HDFS I/O Flow

1. "HDFS bytes read" (MapTask inputs)

When mapper begins, each MapTask reads blocks from HDFS which are mostly locally because of the block's location information.

2. "HDFS bytes write" (ReduceTask ouputs)

At the end of reduce phase, all the results of the MapReduce job will be written to HDFS.

3. "local bytes write/read" (Intermediate inputs/outputs)

All the temporary and intermediate I/O was produced in local Linux file system, mostly happens at the end of the mapper, and at the beginning of the reducer.

- **"local bytes write" scenes**

  a) During MapTask's phase, map's output was firstly written to Buffers, the size and threshold of the Buffers can be configured. When the Buffer size reach the threshold the Spill operation will be triggered. Each Spill operation will generate a temp file "Spill*.out" on disk. And if the map's output was extremely big, it will generate many Spill*.out files, all I/O are "local bytes write".

  b) After MapTask's Spill operation, all the map results are written to disk, the Merge operation begins to combine small spill*.out files to a bigger intermediate.* file. According to Hadoop's algorithm, Merge will be circulating executed until the files' number is less than the configured threshold. During this action it will generate some "local bytes write" when write bigger intermediate.* to disk.

  c) After Merge operation, each map has limited(less than configured threshold) files, including some spill*.out and some intermediate.* files. Hadoop will combine these files to a single file "file.out", each map results only one ouput file "file.out". During this action it will generate some "local bytes write" when write "file.out" to disk.

d) At the beginning of Reduce phase, a shuffle phase is to copy needed map outputs from map nodes. The copy use HTTP. If the shuffle files's size reach the threshold some files will be written to disk which will generate "local bytes write".

e) During reduce-task's shuffle phase, Hadoop will start 2 Merge threads: InMemFSMergeThread and LocalFSMerger. The first one detects the in-memory files number. The second one detects the disk files number, it will be trigged when the disk file number reach threshold to merge small files. LocalFSMerger could generate "local bytes write".

f) After reduce-task's shuffle phase, Hadoop has many copied map results, some in memory and some on disk. Hadoop will merge these small files to a limited (less than threshold) bigger intermediate.* files. The merge operation will generate "local bytes write" when write merged big intermediate.* files to disk.

- **"local bytes read" scenes**
  - a) During MapTask's Merge operation, Hadoop merge small spill*.out files to limited bigger intermediate.* files. During this action it will generate some "local bytes read" when read small spill*.out files from disk.
  - b) After MapTask's Merge operation, each map has limited(less than configured threshold) files, some Spill*.out and some intermediate.* files. Hadoop will combine these files to a single file "file.out". During this action it will generate some "local bytes read" when read spill*.out and intermediate.* files from disk.
  - c) During reduce-task's shuffle phase, LocalFSMerger will generate some "local bytes read" when read small files from disk.
  - d) During reduce-task's phase, all the copied files will read from disk to generate the reduce-task's input. It will generate some "local bytes read".

Differences of each stage between HDFS and Lustre

In this section, we divide Hadoop into 4 stages: mapper input, mapper output, reducer input, reducer output. And we compare the differences of HDFS and Lustre in each stage.

1. Map input: read/write

1) With block's location information
   - *HDFS*: streaming read in each task, most locally, rare remotely network I/O.
   - *Lustre*: parallelly read in each task from Lustre client.

2) Without block's location information
   - *HDFS*: streaming read in each task, most locally, rare remotely network I/O.
   - *Lustre*: parallelly read in each task from Lustre client, less network I/O than the first one because of location information.

Add block's location information will let tasks read/write locally instead of remotely as far as possible. And "add block location information" is for both minimizing the network traffic, and raising read/write speed

2. Map output: read/write

- *HDFS*: writes on local Linux file system, not HDFS, showing in **Figure 4**.
- *Lustre*: writes on Lustre

A record emitted from a map will be serialized into a buffer and metadata will be stored into accounting buffers. When either the serialization buffer or the metadata exceed a threshold, the contents of the buffers will be sorted and written to disk in the background while the map continues to output records. If either buffer fills completely while the **spill** is in progress, the map thread will block. When the map is finished, any remaining records are written to disk and all on-disk segments are merged into a single file. Minimizing the number of spills to disk can decrease map time, but a larger buffer also decreases the memory available to the mapper. Hadoop merge/spill strategy is simple. We think it can be changed to adapt Lustre.

Disperse MapTask intermediate outputs to multiple nodes (on Lustre) can lessen I/O stress of MapTask nodes.

3. **Reduce input: shuffle phase read/write**
   - *HDFS*: Uses HTTP to fetch map output from remote mapper nodes
   - *Lustre*: Build a hardlink of the map output.

   Each reducer fetches the outputs assigned via HTTP into memory and periodically **merges** these outputs to disk. If intermediate compression of map outputs is turned on, each output is decompressed into memory.

   Lustre will delay actual network transmission time (shuffle phase), rather than shuffling all needed map intermediate results at the beginning of ReduceTask. Lustre can disperse the network usage in time scale.

4. **Reduce output: write**
   - *HDFS*: ReduceTask write results to HDFS, each reducer is serial.
   - *Lustre*: ReduceTask write results to Lustre, each reducer can be parallel.

Difference Summary Table

	Lustre	**HDFS**
Host Language	C	Java
POSIX compliance	Support	Not strictly support
Heterogeneous network	Support	Not support, only TCP/IP.
Hard/soft Links	Support	Not support
User/ group quotas	Support	Not support
Access control list (ACL)	Support	Not support
Coherency Model		Write once read many
Record append	Support	Not support
Replication	Not support	Support

Lustre：

1. Lustre does not support Replication.
2. Lustre file system interface fully compliant in accordance with the POSIX standard
3. Lustre support appending-writes to files.
4. Lustre support user quotas or access permissions.
5. Lustre support hard links or soft links.
6. Lustre support Heterogeneous networks, including TCP/IP, Quadrics Elan, many flavors of InfiniBand, and Myrinet GM.

HDFS：

1. HDFS hold stripe replica maintain data integrity. Even if a server fails, no data is lost.
2. HDFS prefers streaming read, not random read.
3. HDFS supports cluster re-balance.
4. HDFS relaxes a few POSIX requirements to enable streaming access to file system data.
5. HDFS applications need a write-once-read-many access model for files
6. HDFS does not support appending-writes to files.
7. HDFS does not yet implement user quotas or access permissions.
8. HDFS does not support hard links or soft links.
9. HDFS does not support Heterogeneous networks, only support TCP/IP network.

Run Hadoop jobs over Lustre

In this section, we give a workable plan for running Hadoop jobs over Lustre. After that, we try to make some improvements for performance, such as: Add block location information for Lustre, Use hardlink[11] in shuffle stage and etc. In the next section, we compare the performance of these improvements by tests.

How to Run MapReduce job with Lustre

Besides HDFS, Hadoop can support some other kinds of file system such as KFS[12], S3[13]. These file systems provided a Java interface which can be used by Hadoop. So Hadoop uses them easily.

Lustre has no JAVA wrapper, it can't adopt like the above file system. Fortunately, Lustre provides a POSIX-compliant UNIX file system interface. We can use Lustre as local file system at each node.

To run Hadoop over Lustre file system, first of all Lustre should installed on every node in the cluster and mounted at the same path such as /Lustre. Modify the configuration which Hadoop used to build the file system. Give the path where Lustre was mounted to the variable 'fs.default.name'. And 'mapred.local.dir' should be set to an independent directory. When running job, just start JobTracker and TaskTracker. In this means, Hadoop will use Lustre file system to store all information.

Details we have done:

1. Mount Lustre at /Lustre on each node;
2. Build a Hadoop-site.xml[14] configuration file and set some variables;
3. Start JobTracker and TaskTracker;
4. Now we can run MapRedcue jobs over Lustre.

Add Lustre block location information

As it is mentioned before, "add block location information" is for both minimizing the network traffic, and raising read/write speed. HDFS maintains the block location information, which makes most of Hadoop MapTasks can be assigned to the node where it can read/write locally. Surely, reading from local disk has a better performance than reading remotely when the cluster network is busy or slow. In this situation, HDFS can reduce the network traffic and short map phase. Even if the cluster has excellent network, let map task reading locally can reduce network traffic significantly.

In order to improve the performance of Hadoop using Lustre as storage backend, we have to help JobTracker to access the data location of each split. Lustre provides some user utility to gather the extended attributes of a specific file. Because the Hadoop default block size is 32M, we set Lustre stripe size to 32M. This will make that each map task input is on a single node.

We create a file containing the map from OST->hostname. This can be done with "lctl dl" command at each node. Another way, we add a new JAVA class to Hadoop source code and rebuild it to get a new jar. Through the new class we can get location information of each file stored in Lustre. The location information is saved in an array. When JobTracker pre-assign map tasks, these information helps Hadoop to give map task to the node(OSS) where it can read input from local disk.

Details of each step of adding block location info:

First, figure out task node's hostname from OST number (lfs getstripe) and save as a configuration file.

For example:

```
Lustre-OST0000_UUID=oss2
Lustre-OST0001_UUID=oss3
Lustre-OST0002_UUID=oss1
```

Second, get location information of each file using function in Jshell.java. The location information will save in an array.

Third, Modify FileSystem.java. Before modify getFileBlockLocations in FileSystem.java simply return an elt containing 'localhost'.

Four, we will attach location information to each block (stripe).

Differences of BlockLocation[] *(FileSystem. getFileBlockLocations())* between before and after:

	hosts	names	offset	length
Before Modify	localhost	localhost:50010	offset in file	length of block
After Modify	hostname	hostname:50010	offset in file	length of block

Input/Output Formats

There are many input/output formats in Hadoop, such as: SequenceFile*(Input/Output)*Format, Text*(Input/Output)*Format, DB*(Input/Output)*Format, etc. We can also implement our own file format by implementing InputFormat/OutputFormat interfaces. In this section, we firstly what typical input/output formats are and how they worked. At last, we explain the SequenceFile format in detail to bring a more clearly understanding.

Format Inheritance UML

Figure 5, 6 shows the Inheritance UML of Hadoop format.

- InputFormat logically split the set of input files for the job. The Map-Reduce framework relies on the InputFormat of the job to:

1. Validate the input-specification of the job.
2. ***getSplits()***:Split-up the input file(s) into logical InputSplits, each of which is then assigned to an individual Mapper.
3. ***getRecordReader():*** Provide the RecordReader implementation to be used to glean input records from the logical InputSplit for processing by the Mapper.

■ OutputFormat describes the output-specification for a Map-Reduce job. The Map-Reduce framework relies on the OutputFormat of the job to:

1. ***checkOutputSpecs():*** Validate the output-specification of the job. For e.g. check that the output directory doesn't already exist.
2. ***getRecordWriter():*** Provide the RecordWriter implementation to be used to write out the output files of the job. Output files are stored in a FileSystem.

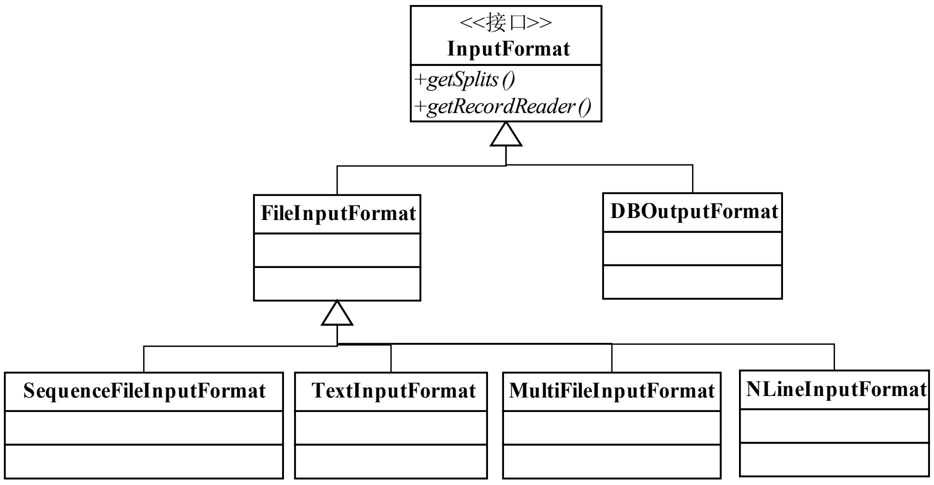


Figure 5: InputFormat Inheritance UML

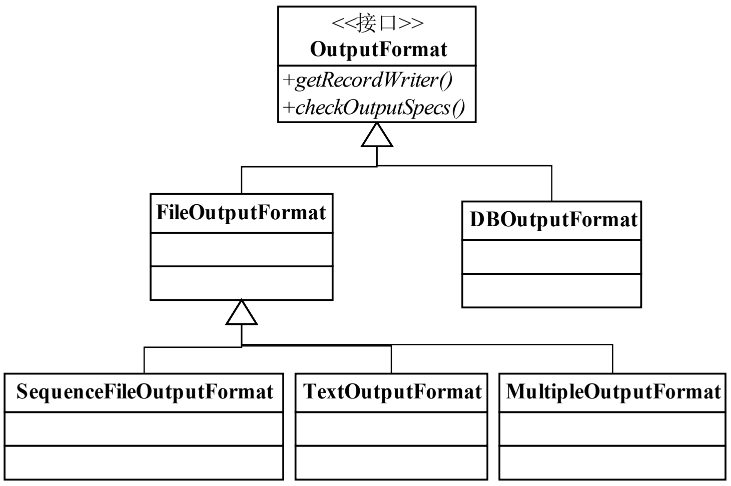


Figure 6: OutputFormat Inheritance UML

SequenceFileInputFormat/SequenceFileOutputFormat

This is An InputFormat/OutputFormat SequenceFiles. This format is used in BigMapOutput test.

SequenceFileInputFormat

✓ ***getSplits()***: splits files by file size and user configuration, and Splits files returned by #listStatus(JobConf) when they're too big..

- ✓ ***getRecordReader() :***return SequenceFileRecordReader, a decoration of SequenceFile.Reader, which can seek to the needed record(key/value pair), read the record one by one.

SequenceFileOutputFormat

- ✓ ***checkOutputSpecs():***Validate the output-specification of the job.
- ✓ ***getRecordWriter():*** return RecordReader, a decoration of SequenceFile.Writer, which use append() to write record one by one.

SequenceFile physical format

SequenceFiles are flat files consisting of binary key/value pairs. SequenceFile provides Writer, Reader and Sorter classes for writing, reading and sorting respectively.

There are three SequenceFile Writers based on the CompressionType used to compress key/value pairs:

1. Writer : Uncompressed records.
2. RecordCompressWriter : Record-compressed files, only compress values.
3. BlockCompressWriter : Block-compressed files, both keys & values are collected in 'blocks' separately and compressed. The size of the 'block' is configurable.

The actual compression algorithm used to compress key and/or values can be specified by using the appropriate CompressionCodec. The recommended way is to use the static createWriter methods provided by the SequenceFile to choose the preferred format. The Reader acts as the bridge and can read any of the above SequenceFile formats.

Essentially there are 3 different formats for SequenceFiles depending on the CompressionType specified. All of them share a common header described below.

SequenceFile Header

- version - 3 bytes of magic header **SEQ**, followed by 1 byte of actual version number (e.g. SEQ4 or SEQ6)
- keyClassName -key class
- valueClassName - value class
- compression - A boolean which specifies if compression is turned on for keys/values in this file.
- blockCompression - A boolean which specifies if block-compression is turned on for keys/values in this file.
- compression codec - CompressionCodec class which is used for compression of keys and/ or values (if compression is enabled).
- metadata - Metadata for this file.
- sync - A sync marker to denote end of the header.

Uncompressed SequenceFile Format

- Header
- Record
  - Record length
  - Key length
  - Key
  - Value
- A sync-marker every few 100 bytes or so.

Record-Compressed SequenceFile Format

- Header
- Record
    - Record length
    - Key length
    - Key
    - *Compressed* Value
- A sync-marker every few 100 bytes or so.

Block-Compressed SequenceFile Format

- Header
- Record *Block*
    - Compressed key-lengths block-size
    - Compressed key-lengths block
    - Compressed keys block-size
    - Compressed keys block
    - Compressed value-lengths block-size
    - Compressed value-lengths block
    - Compressed values block-size
    - Compressed values block
- A sync-marker every few 100 bytes or so.

The compressed blocks of key lengths and value lengths consist of the actual lengths of individual keys/values encoded in ZeroCompressedInteger format.

Test cases and test results

In this section, we introduce our test environment, and then we'll describe three kinds of tests: Application test, sub phases test, and benchmark I/O test.

Test environment

Neissie：

Our experiments run on cluster with 8 nodes in total, one is mds/namenode, the rest are OSS/DataNode. Every node has the same hardware configuration with two 2.2 GHz processors, 8G of memory, and Gigabit Ethernet.

Application Tests

There are two kinds of typical applications or tests which must deal with huge inputs: General statistic applications (such as: *WordCount*[J]), Computational complexity applications (such as: *BigMapOutput*[9], *webpages analytics processing*).

✧ *WordCount:*

This is a typical test case in Hadoop. It reads input files which must be text files and then count how often each words occur. The output is also text files, each line of which contains a word and the count of times it occurred, separated by a tab.

Why we need this: As we described before, for some applications which have small MapOutput , MapReduce+HDFS woks well. And the WordCount can represent this kind of applications, it may have big input files, but either the MapOutput files or reduce output files is small, and just do some statistics work. Through this test we can see the performance gap between Lustre and HDFS for statistics applications.

✧ *BigMapOutput:*

It is a map/reduce program that works on a very big non-splittable file , for map or reduce tasks it just read the input file and the do nothing but output the same file.

Why we need this: WordCount represents some applications which have small output file, but BigMapOutput is just the opposite. This test case generates big output, and as mentioned previous when running this kind of applications, MapReduce+HDFS will have a lot of disadvantages. Through this test we should see Lustre has better performance than HDFS, especially after the changes we had made to Hadoop, that is to say Lustre is more effective than HDFS for this kind of applications.

Test1: WordCount with a big file

In this test, we choose the WordCount example(details as before), and process one big text file(6G). Set the blocksize=32m(actually all the tests have the same blocksize), so we would get about 200 Map tasks in all, and base on the mapred tutorial's suggestion, we set the Reduce Tasks=0.95*2*7=13.

To get better performance we have tried several Lustre client's configuration(details as follows).

It should be noted that we didn't make any changes to this test.

Result:

Item	*HDFS*	*Lustre*	*Lustre*	*Lustre*
Configuration	*default*	*All directory* *Stripesize=1M* *Stripecount=7*	*Input directory* *Stripesize=32M* *Stripecount=7* *Other directories* *Stripesize=1M* *Stripecount=7*	*Input directory* *Stripesize=32M* *Stripecount=7* *Other directories* *Stripesize=1M* *Stripecount=1*
Time (s)	*303*	*325*	*330*	*318*

Test2:WordCount with many small files

In this test, we choose the WordCount example(details as before), and process a large number small files(10000). we set the Reduce Tasks=0.95*2*7=13.

It should be noted that we didn't make any changes to this test.

Result:

Item	*HDFS*	*Lustre*
Configuration	*default*	*Stripesize=1M*

		Stripecount=1 *Stripeindex=-1*
Time	*1h19m16s*	1h21m8s

Test3: BigMapOutput with one big file

In this test, the BigMapOutput process a big Sequence File (this kind of file format is defined in Hadoop, the size is 6G).

It should be noted that we had made some changes to this test to compare HDFS with Lustre. We will describe these changes in detail in the follow.

Result1:

In this test, we changed the input file format so the job can launch may Map tasks(every Map task process about 32m data,and the default situation is that just one Map task process the whole input file),and launch 13 Reduce Tasks. otherwise the mapred.local.dir is set on Lustre(the default value is /tmp/Hadoop-${user.name}/mapred/local, in other words it is on the local file system).

Item	HDFS	Lustre stripesize=1M stripecount=7
Time (s)	164	392

Result2:

Because every node in the cluster has large memory size, so there are many operations finished in the memory. So in this test, we made just one change to force some data must be spilled to the disk, of course that means more pressure to the file system.

Item	HDFS	Lustre stripesize=1M stripecount=7
Time (s)	207	446

Result3:

From the results as before, we find that HDFS's performance much higher than Lustre's, which was unexpected. We thought the reason was os cache. To prove it we set the mapred.local.dir to default value same as HDFS, and the result show the guess is right.

Item	HDFS	Lustre stripesize=1M stripecount=7
Time (s)	207	207

Sub phases tests

As we discussed, Hadoop has 3 stages I/O: mapper input, copy mapper output to reducer input, reducer output. In this section, we make tests in these sub phases to find the actually time-consuming phase in each specific application.

Test4: BigMapOutput with hardlink

In this test, we still choose BigMapOutput example, but the difference is that in copy map_output phase, we create a hardlink to the map_output file to avoid copying output file from one node to another on Lustre, after this change we can expect better performance about Lustre with hardlink way.

Result:

Item	Lustre	Lustre with hardlink
Time (s)	446	391

Lustre: stripesize=1M, stripecount=7

Test5: BigMapOutput with hardlink and location information

In this test, we export the data location information on Lustre to mapred tasks through changing the Hadoop code and create a hardlink in copy map_output phase. But in our test environment, the performance has no significant difference between local read and remote read, because of the high write speed.

Result:

Item	Lustre with hardlink	Lustre with hardlink location info
Time (s)	391	366

Test6: BigMapOutput Map_Read phase

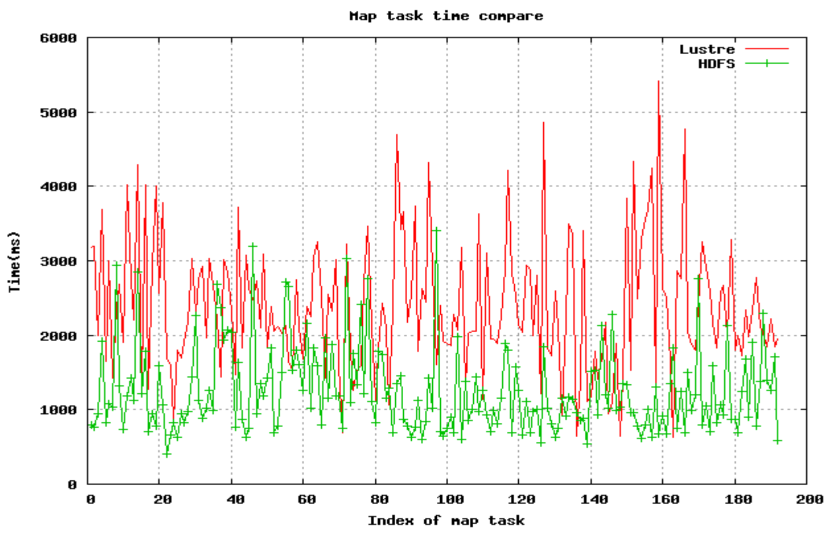
In this test, we let the Map tasks just read input file but do nothing, so we can see the difference of read performance more clearly.

Results:

Item	HDFS	Lustre	Lustre location info
Time (s)	66	111	106

Lustre: stripesize=32M, stripecount=7

In order to find out why HDFS get a better performance, we tried to analyst the log files. We have scanned all map task logs, and find out cost time of reading input of each task.



Test7: BigMapOutput copy_MapOutput phase

In this test, we will compare shuffle phase(copy MapOutput phase) between Lustre with hardlink and Lustre without hardlink, and this time we set the Reduce Task num to 1, so we can see the difference clearly.

Result:

Item	Lustre with hardlink	Lustre
Time (s)	228	287

Lustre: stripesize=1M, stripecount=7

Test8: BigMapOutput Reduce output phase

In this test, we just compare the Reduce output phase, and we set the Reduce Task num to 1 the same as above.

Result:

Item	HDFS	Lustre
Time (s)	336	197

Lustre: stripesize=1m, stripecount=7

Benchmark I/O tests

1. **IOR tests**

◆ **IOR on HDFS**

Setup: 1namenode 2datanode

Note:

1. We have used "-Y" to perform fsync after each POSIX write.
2. We also separate the write and read tests.

Operation	Client	Block size	Aggregate filesize	Speed
write	1	2G	2G	32.28 M/s
write	2	2G	4G	23.34 M/s
write	3	2G	6G	36.43 M/s
read	1	2G	2G	8.72 M/s
read	2	2G	4G	5.72 M/s

◆ **IOR on Lustre**

Setup: 1mds 2oss 2ost per OSS.

Note:

We have a dir named 2stripe whose stripe count is 2 and stripe size is 1M.

Operation	Client	Block size	Aggregate filesize	Speed
write	1	2G	2G	12.02M/s
write	2	2G	4G	18.02M/s
write	3	2G	6G	31.07M/s
read	1	2G	2G	13.42M/s
read	2	2G	4G	15.66M/s
read	3	2G	6G	22.08M/s

2. **Iozone**

◆ **Iozone on HDFS**

Setup: 1namenode 2datanode

Note:

1. 16M is the biggest record size in our system (which should less than the maximum buffer size in our system).
2. We have used "-U" to unmount and mount between tests, this guarantees that the buffer cache doesn't contain the file under test.
3. We also have used "-p" to purge the processor cache before every test.
4. We have tested the write and read performance separately to ensure that the result will not be affected by the client cache.

Operation	File size	Rec size	Speed
write	2G	16M	48.823M/s
rewrite	2G	16M	38.484M/s
read	2G	16M	51.886M/s
reread	2G	16M	51.718M/s

◆ **Iozone on Lustre**

Setup 1: 1mds 1oss 2ost

stripe count=2 stripe size=1M

Operation	File size	Rec size	Speed
write	2G	16M	11.849M/s
rewrite	2G	16M	11.842M/s
read	2G	16M	11.429M/s
reread	2G	16M	11.475M/s

Setup 2: 1mds 2oss 4ost

stripe count=4 stripe size=1M

Operation	File size	Rec size	Speed
write	2G	16M	11.854M/s
rewrite	2G	16M	12.120M/s
read	2G	16M	11.470M/s
reread	2G	16M	11.453M/s

Setup 3: 1mds 2oss 6ost

stripe count=6 stripe size= 1M

Operation	File size	Rec size	Speed
write	2G	16M	12.552M/s
rewrite	2G	16M	12.086M/s
read	2G	16M	11.463M/s
reread	2G	16M	11.455M/s

References and glossaries

[1] HDFS is the primary distributed storage used by Hadoop applications. A HDFS cluster primarily consists of a NameNode that manages the file system metadata and DataNodes that store the actual data. The HDFS Architecture describes HDFS in detail.

[2] MapReduce: http://labs.google.com/papers/mapreduce-osdi04.pdf

[3] Hadoop: http://Hadoop.apache.org/core/docs/r0.20.0/mapred_tutorial.html

[4] JobTracker :http://wiki.apache.org/Hadoop/JobTracker

[5] TaskTracker :http://wiki.apache.org/Hadoop/TaskTracker

[6] A interface that provides Hadoop Map() function, also refer MapTask node;

[7] A interface that provides Hadoop Reduce() function, also refer ReduceTask node;

[8] WordCount: A simple application that counts the number of occurrences of each word in a given input set.
http://Hadoop.apache.org/core/docs/r0.20.0/mapred_tutorial.html#Example%3A+WordCount+v2.0

[9] BigMapOutput: A Hadoop MapReduce test whose intermediate results are huge.
https://svn.apache.org/repos/asf/Hadoop/core/branches/branch-0.15/src/test/org/apache/Hadoop/mapred/BigMapOutput.java

[10] **merge/spill operations:** A record emitted from a map will be serialized into a buffer and metadata will be stored into accounting buffers. When either the serialization buffer or the metadata exceed a threshold, the contents of the buffers will be sorted and written to disk in the background while the map continues to output records. If either buffer fills completely while the **spill** is in progress, the map thread will block. When the map is finished, any remaining records are written to disk and all on-disk segments are merged into a single file. Minimizing the number of spills to disk can decrease map time, but a larger buffer also decreases the memory available to the mapper. Each reducer fetches the output assigned via HTTP into memory and periodically **merges** these outputs to disk. If intermediate compression of map outputs is turned on, each output is decompressed into memory.

[11]

[12] Kosmos File System: http://kosmosfs.sourceforge.net/

[13] Amazon S3 is storage for the Internet. It is designed to make web-scale computing easier for developers. : http://aws.amazon.com/s3/

[14] Lustre’s Hadoop-site.xml is in attachment file.