工信部测试经验总结

测试时间：6 月 10 日-6 月 17 日

测试人员：陆扬，随志浩，冯永设，严东荣，张永曦

测试条目：

功能测试：Namenode HA，HMaster HA，Datanode 及 RegionServer 节点失效并恢复

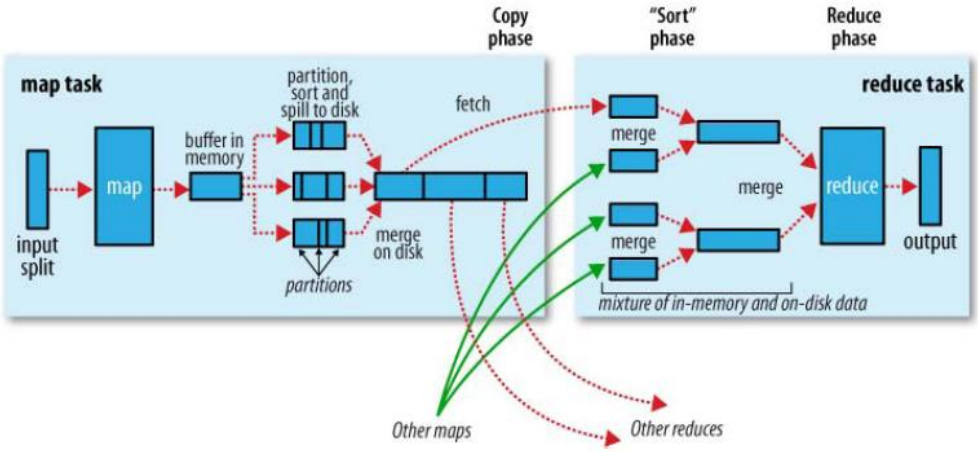
安全性测试：存储加密（Hive 及 HBase），身份认证，统一用户管理，权限管理（HDFS,Hive 及 HBase）

性能测试：Hive(Join,Aggregation)，HBase（Write,Scan,Read）,Hadoop(Wordcout,TeraSort, PageRank,Kmeans 及 Naive Bayes)

1、Hadoop 性能调优

1.1 MR 性能影响因素

MR 执行流程分析，探讨 MR 性能影响参数（执行流程+核心参数）



Map 的执行结果先放到内存中，内存不够时，通过 spill 过程持久化到硬盘中，在 spill 之前要对中间结果进行排序：partition->sort->spill

过程 1：Map 数据输入及切片

Map 任务提交到 Yarn 后，被 ApplicationMaster 启动，任务的形式是 YarnChild，在其中会执行 MapTask 的 run 方法。在提交之前，JobClient 会对数据源进行切片，切片信息由 InputSplit 对象封装，接口定义如下：

JobClient 通过 getSplits 方法来计算切片信息，切片默认大小和 HDFS 的块大小相同。JobSplitWriter 将 Splits 信息（SplitMetaInfo），写入任务执行目录的文件中，SplitMetaInfo 保

存了该 Split 的数据大小及数据所在的位置。

可以通过配置参数（mapreduce.input.fileinputformat.split.minsize），指定 Map 处理数据量的大小，从而控制 Map 数目，该数目决定了 Map 的执行并行度(Average Map Time)。

MapTask.run，根据 splitIndex 从 SplitsMetaInfo 文件中获取对应的 InputSplit 信息，然后创建 RecordReader，从数据文件中获取 Key 及 Value。然后调用 Mapper.run，执行定义的操作。

过程 2：Map 数据处理

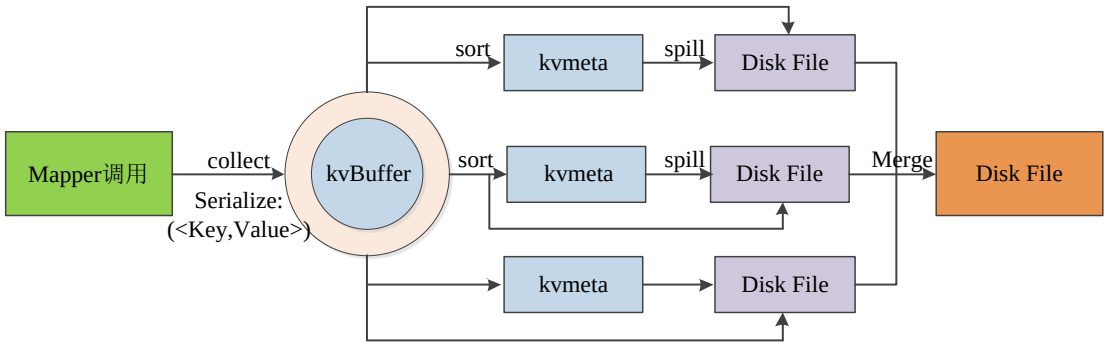
Mapper.run 的接口源码如下所示：

```
1·   public void run(Context context) throws IOException, InterruptedException {
2·     setup(context);
3·     try {
4·       while (context.nextKeyValue()) {
5·         map(context.getCurrentKey(), context.getCurrentValue(), context);
6·       }
7·     } finally {
8·       cleanup(context);
9·     }
```

map 方法，会调用 MapOutputCollector.collect(Key,Value)，输出到内存中。

过程 3：Map Sort 及 Spill

Mapper.run 在执行过程中，将处理结果写到内存缓冲区中，当缓冲区被使用完后，需要 spill 到磁盘中，过程如下图所示：



MapOutputCollector#collect，将<Key,Value>序列化后，存储到内存缓冲区 kvBuffer 中，kvBuffer 是内存中的一个环形数据结构，当 Map 输出的数据大小超过 KvBuffer 配置的大小，会调用 startSpill 方法进行溢写。KvBuffer 缓冲区的大小通过参数 mapreduce.task.io.sort.mb 进行配置，这个值越大，溢出到磁盘次数越少，减少 Map 端的 I/O 时间，但是增加这个值会导致每个 Map 任务需要的内存增加（可能会撑爆内存）。

为了提高 IO 的利用率，可使用压缩，由参数：mapreduce.output.fileoutputformat.compress.codec 及 mapreduce.map.output.compress.codec 配置输出是否压缩。

在 spill 到磁盘之前，要进行排序操作，Sorter.sort 对 kvBuffer 中的数据进行排序，排序

的时间和缓冲区大小相关。

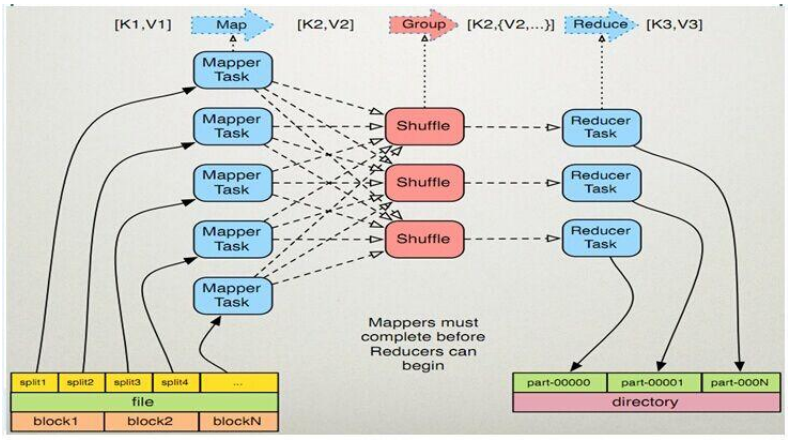
map 执行结束后，spill 到磁盘的文件一般为多个，需要对文件进行 merge 操作，操作的过程中会对文件中的数据进行归并，归并所花费的时间，与 io.sort.mb 的大小配置相关（Average merge Time）。

这个过程结束后，每个 Map 会生成一个 MapOutputFile，Map 结束后通知 AM，启动 Reduce 任务。

过程 4：Reduce Shuffle

Map 结束后，Reduce 启动 shuffle 阶段，从 Map 输出文件中获取数据，但是 Map 一般多个，某个 Reduce 从多个 Map 输出文件中获取其所对应的处理数据。MR 提供 Partitioner 接口，作用是根据 Key 或 Value 及 Reduce 的数量来决定 Collect 输出的数据由哪个 Reduce Task 处理，默认是 Key Hash 后再以 Reduce Task 数量取模。

过程如下图所以：



Reduce 分为 copy（shuffle）->sort->reduce 三个阶段，在 Reduce 进行数据之前，都进行数据的拉取（copy）。Map 数目为 M，Reduce 数目 N，则需要建立 Fetcher 数目 M*N。

而且这个过程涉及到的大量的 IO 操作(Average Shuffle Time)。copy 完毕后，Reduce 会对拉取的数据进行 merge 操作（包括排序）。

Average Shuffle Time，是 Map Reduce 很重要的影响因素。

过程 5：Reduce 数据处理

在上一步的 Shuffle 获取<Key,Value>的 RawKeyValueIterator，在使用之前要进行反序列化，核心方法是 ReduceContextImpl#nextKeyValue:

```
@Override
public boolean nextKeyValue() throws IOException, InterruptedException {
  if (!hasMore) {
    key = null;
    value = null;
    return false;
  }
  firstValue = !nextKeyIsSame;
  DataInputBuffer nextKey = input.getKey();
  currentRawKey.set(nextKey.getData(), nextKey.getPosition(),
                    nextKey.getLength() - nextKey.getPosition());
  buffer.reset(currentRawKey.getBytes(), 0, currentRawKey.getLength());
  key = keyDeserializer.deserialize(key);
  DataInputBuffer nextVal = input.getValue();
  buffer.reset(nextVal.getData(), nextVal.getPosition(), nextVal.getLength()
      - nextVal.getPosition());
  value = valueDeserializer.deserialize(value);
```

反序列化后，数据交由 Reduce 来处理，Reducer 源码的接口，如下所示：

```
1·  while (context·nextKey()) {
2·          reduce(context·getCurrentKey(), context·getValues(), context);
3·          // If a back up store is used, reset it
4·          Iterator<VALUEIN> iter = context·getValues()·iterator();
5·          if(iter instanceof ReduceContext·ValueIterator) {
6·            ((ReduceContext·ValueIterator<VALUEIN>)iter)·resetBackupStore(); }}
```

reduce 方法中使用 ReduceContex#write 写入到输出文件中（Average Reduce Time）。

总结：MR 的执行及影响因素

影响因素 1，Map 及 Reduce Task 数目

执行阶段	影响因素	配置参数及默认值	说明
Map 数据处理	Map 处理数据量	fileinputformat.split.minsize:0 dfs.blocksize：128M	计算公式： max(minSize,Math.min(maxSize, blockSize)
MapOutput	排序，spill merge	mapreduce.task.io.sort.mb:100M map.output.compress.codec:DefaultCodec	主要耗费时间：KvBuffer -> Disk Sort 及 Merge Time
Reduce Shuffle	copy,merge	mapreduce.reduce.shuffle.parallelcopies：5 mapreduce.job.reduces：1	Map Num， Map output 数据量 Reduce 数目
Reduce 计算	数据量		

MR 计算影响因素 2：系统资源，主要：cpu 及内存

yarn.nodemanager.resource.memory-mb	
yarn.nodemanager.resource.cpu-vcore	
mapreduce.map.cpu.vcores	
mapreduce.reduce.cpu.vcores	
mapreduce.map.memory.mb	
mapreduce.reduce.memory.mb	
mapreduce.task.io.sort.mb	
mapred.child.java.opts	
mapreduce.reduce.java.opts	
mapreduce.map.java.opts	
yarn.scheduler.maximum-allocation-mb	

MR 影响因素 3：

dfs.domain.socket.path	/var/lib/hadoop-hdfs/dn_socket
mapreduce.job.map.output.collector.class	org.apache.hadoop.mapred.nativetask.NativeMapOutputCollectorDelegator

1.2 MR 内存配置

内存是 MR Task 运行过程中很难处理的资源，要考虑的因素很多

1）系统可提供内存

2）Server 预留内存

3）Map 及 Reduce Task 需要内存（与初始 Map 处理数据量和运算类型相关）

4）参数的配置关系

相关的参数如下表：

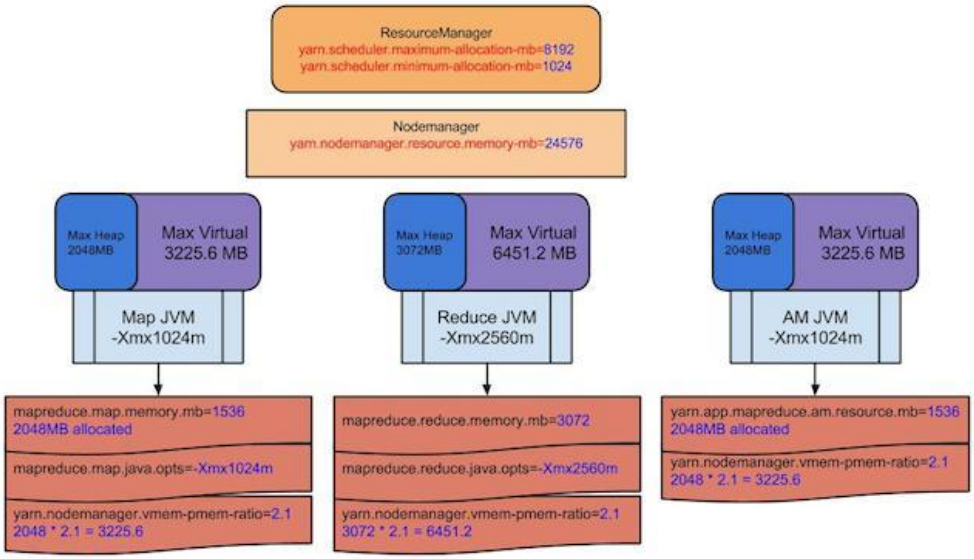
参数	说明
yarn.nodemanager.resource.memory-mb	节点上 YARN 可使用的物理内存总量
yarn.scheduler.maximum-allocation-mb	单个任务可申请的最多物理内存量
yarn.scheduler.minimum-allocation-mb	单个任务可申请的最少物理内存量
yarn.nodemanager.vmem-pmem-ratio	任务每使用 1MB 物理内存， 最多可使用虚拟内存量，默认是 2.1
mapreduce.map.memory.mb	每个 Map Task 所需要的内存量
mapreduce.reduce.memory.mb	每个 Reduce Task 所需要的内存量
mapred.child.java.opts	Task 的 JVM 堆大小
mapreduce.map.java.opts	Map Task 的 JVM 堆大小（真正用于工作的堆容量）
mapreduce.reduce.java.opts	Reduce Task 的 JVM 堆大小
mapreduce.job.heap.memory-mb.ratio	JVM opts/Container Java Size （默认 0.8）

JVM 堆大小与 Task 所需要的内存量的公式如下所示：

mapreduce.{map|reduce}(child).java.opts=mapreduce.{map|reduce}.memory.mb * mapreduce.heap.memory-mb.ratio

Map 的并发数量(需要不大于 CPU 核数)×mapred.child.java.opts < NM 总内存

下图是内存配置的一个例子：



参数	说明
yarn.nodemanager.resource.memory-mb	24GB
yarn.scheduler.maximum-allocation-mb	8GB
yarn.scheduler.minimum-allocation-mb	1GB
yarn.nodemanager.vmem-pmem-ratio	2.1
mapreduce.map.memory.mb	1.5G(逻辑分配 2GB)
mapreduce.reduce.memory.mb	3G
yarn.app.mapreduce.am.resource.mb	1.5G(逻辑分配 2GB)
mapreduce.map.java.opts	-Xmx1024M(1G)
mapreduce.reduce.java.opts	-Xmx2560M (2.5G)
mapreduce.job.heap.memory-mb.ratio	JVM opts/Container Java Size （默认 0.8）

如上图所示：

Map Container 请求 1.5GB 内存，AM Container 请求 1.5GB，由于 yarn.scheduler.minimum-allocation-mb 设置为 1GB，因此 Map 及 AM 实际分配的内存是 2G(Logical Allocation)。

mapreduce.map.java.opts=-Xmx1024m，该参数适合与 Logical Allocation 2G 的情形，如果 Map 使用内存超过 2GB,NM 会 kill 掉该 Task，并抛出异常：

> *Current usage: 2.1gb of 2.0gb physical memory used; 1.6gb of 3.15gb virtual memory used. Killing container.*

虚拟内存：yarn.nodemanager.vmem-pmem-ratio 默认是 2.1，表明 Map 或者 Reduce 可以使用 mapreduce.reduce.memory.mb 或者 mapreduce.map.memory.mb 配置内存的 2.1 倍。

Java 堆的设置，应该比 Hadoop Container Memory 配置的要小，因为需要为 Java Code 保存一定的内存，建议是保留 20%的内存。

Mapper 是 Java 进程，每个 JVM 的分配的堆大小是通过 mapred.map.java.opts（或者 mapreduce.reduce.java.opts）来配置的。如果 Mapper 进程耗尽所有 Heap Memory，会抛出：

```
Error: java.lang.RuntimeException: java.lang.OutOfMemoryError
```

在 Hadoop 集群中，要均衡内存、cpu 和磁盘的使用，尽量不让某种资源成为限制。下面介绍如何计算内存：

1）确认节点的以下资源，RAM、CPU 核数及 Disks 的盘数（一般建议每块盘两个 Container）

2）计算预留 Memory，系统进程及其他 Hadoop 服务的进程，例如 HBase，下面是建议的预留资源

Total Memory per Node	Recommended Reserved System Memory	Recommended Reserved HBase Memory
4 GB	1 GB	1 GB
8 GB	2 GB	1 GB
16 GB	2 GB	2 GB
24 GB	4 GB	4 GB
48 GB	6 GB	8 GB
64 GB	8 GB	8 GB
72 GB	8 GB	8 GB
96 GB	12 GB	16 GB
128 GB	24 GB	24 GB
256 GB	32 GB	32 GB
512 GB	64 GB	64 GB

3）计算每个节点可运行的最大 containers 数目，可以使用下面的公式：

```
Containers = minimum of (2*CORES, 1.8*DISKS, (Total available RAM) / MIN_CONTAINER_SIZE)
```

MIN_CONTAINER_SIZE 是最小的 container RAM，这个值与节点可用内存有关，下面是推荐的设置：

Total RAM per Node	Recommended Minimum Container Size
Less than 4 GB	256 MB
Between 4 GB and 8 GB	512 MB
Between 8 GB and 24 GB	1024 MB
Above 24 GB	2048 MB

4）计算每个 Container 的 RAM 总量

```
RAM-per-Container = maximum of (MIN_CONTAINER_SIZE, (Total Available RAM) / Containers))
```

5）计算后，YARN 及 MapReduce 到的配置，如下进行设置

Configuration File	Configuration Setting	Value Calculation
yarn-site.xml	yarn.nodemanager.resource.memory-mb	= Containers * RAM-per-Container
yarn-site.xml	yarn.scheduler.minimum-allocation-mb	= RAM-per-Container
yarn-site.xml	yarn.scheduler.maximum-allocation-mb	= containers * RAM-per-Container
mapred-site.xml	mapreduce.map.memory.mb	= RAM-per-Container
mapred-site.xml	mapreduce.reduce.memory.mb	= 2 * RAM-per-Container
mapred-site.xml	mapreduce.map.java.opts	= 0.8 * RAM-per-Container
mapred-site.xml	mapreduce.reduce.java.opts	= 0.8 * 2 * RAM-per-Container
yarn-site.xml (check)	yarn.app.mapreduce.am.resource.mb	= 2 * RAM-per-Container
yarn-site.xml (check)	yarn.app.mapreduce.am.command-opts	= 0.8 * 2 * RAM-per-Container

6）举例如下，集群节点中 CPU 核数 12,48GB 内存及 12 个磁盘，计算如下：

Reserved Memory = 6GB

Min Container Size = 2G

Containers = minimum of(2*12,1.8*12,(48-6)/2) = 21

RAM-per-Container = maximum of (2,(48-6)/21) = 2

则，配置如下：

Configuration	Value Calculation
yarn.nodemanager.resource.memory-mb	= 21 * 2 = 42*1024 MB
yarn.scheduler.minimum-allocation-mb	= 2*1024 MB
yarn.scheduler.maximum-allocation-mb	= 21 * 2 = 42*1024 MB
mapreduce.map.memory.mb	= 2*1024 MB
mapreduce.reduce.memory.mb	= 2 * 2 = 4*1024 MB
mapreduce.map.java.opts	= 0.8 * 2 = 1.6*1024 MB
mapreduce.reduce.java.opts	= 0.8 * 2 * 2 = 3.2*1024 MB
yarn.app.mapreduce.am.resource.mb	= 2 * 2 = 4*1024 MB
yarn.app.mapreduce.am.command-opts	= 0.8 * 2 * 2 = 3.2*1024 MB

7）配置计算脚本 ，yarn-uti.py，命令如下

```
python yarn-util.py <options>
```

Option	Description
-c CORES	The number of cores on each host.
-m MEMORY	The amount of memory on each host in GB.
-d DISKS	The number of disks on each host.
-k HBASE	"True" if HBase is installed, "False" if not.

例如：

```
python yarn-utils.py -c 16 -m 64 -d 4 -k True
```

返回的配置如下：

Using cores=16 memory=64GB disks=4 hbase=True
Profile: cores=16 memory=64GB reserved=16GB usableMem=48GB disks=4
Num Container=32
Container Ram=1536MB
Used Ram=48GB
Unused Ram=16GB
yarn.scheduler.minimum-allocation-mb=1536
yarn.scheduler.maximum-allocation-mb=49152
yarn.nodemanager.resource.memory-mb=49152
mapreduce.map.memory.mb=1536
mapreduce.map.java.opts=-Xmx1228m
mapreduce.reduce.memory.mb=3072
mapreduce.reduce.java.opts=-Xmx2457m
yarn.app.mapreduce.am.resource.mb=3072
yarn.app.mapreduce.am.command-opts=-Xmx2457m
mapreduce.task.io.sort.mb=614

工信部测试内存分配：

1. 服务器型号是戴尔的 R730，服务器总数是 16 台
2. CPU：2*英特尔至强 E5-2620 v3 2.4GHz,15M 缓存， 12 核（超线程 x2）
3. 内存 8*8GB RDIMM, 2133 MT/s； 64GB
4. 硬盘 10*1.2TB 10K RPM SAS 6Gbps 2.5 英寸 热插拔硬盘 10 块盘
5. 网卡是 2 口万兆网卡，交换机是华三万兆交换机，只用 1 个口

公式计算与实际分配如下：

Configuration	Value Calculation
yarn.nodemanager.resource.memory-mb	50GB(48GB)
yarn.scheduler.minimum-allocation-mb	1GB (1.5GB)
yarn.scheduler.maximum-allocation-mb	8GB(48GB)
mapreduce.map.memory.mb	2GB(1.5GB)
mapreduce.reduce.memory.mb	4GB(3GB)
mapreduce.map.java.opts	1GB(1.6GB)
mapreduce.reduce.java.opts	1GB(3.2GB)
yarn.app.mapreduce.am.resource.mb	2GB(4GB)
yarn.app.mapreduce.am.command-opts	2GB(3.2GB)

注：

实际的配置有以下问题，可用内存有 2GB 浪费（48VCore），java.opts 分配太小（container logica allocation 内存不能合理使用），AM 分配内存较小（则测试中出现过 AM 死掉的情况）

mapreduce.task.io.sort.mb 缓冲区大小，过小对性能有影响，太大影响 Map 的执行，和 Task 分配的内存和堆大小相关，一般是堆大小的 50%

https://support.pivotal.io/hc/en-us/articles/201462036-Mapreduce-YARN-Memory-Parameters

1.3 Map 及 Reduce 数目设置

思路：

控制 Map 个数，减少调度开销

合理设置 Reduce 个数，避免单 Task 数据量处理过大或者过少

1）Map 数量通常是由 Hadoop 集群的 DFS 块大小确定的，输入文件所块数。并行度大致规模是每个 Node 是 10-100 个，CPU 消耗较小的 Map 数量可适当增加，比较合理的情况是每个 Map 的执行时间至少 1 分钟（Task 的启动时间开销较大，调度开销 1-5s）。

Map 数量可由 mapreduce.input.fileinputformat.split.minsize 和 mapred.map.tasks 设置，mapred.map.tasks 只有在 InputFormat 决定的 map 任务个数值小时才起总用。

组内资源不足限制并行度

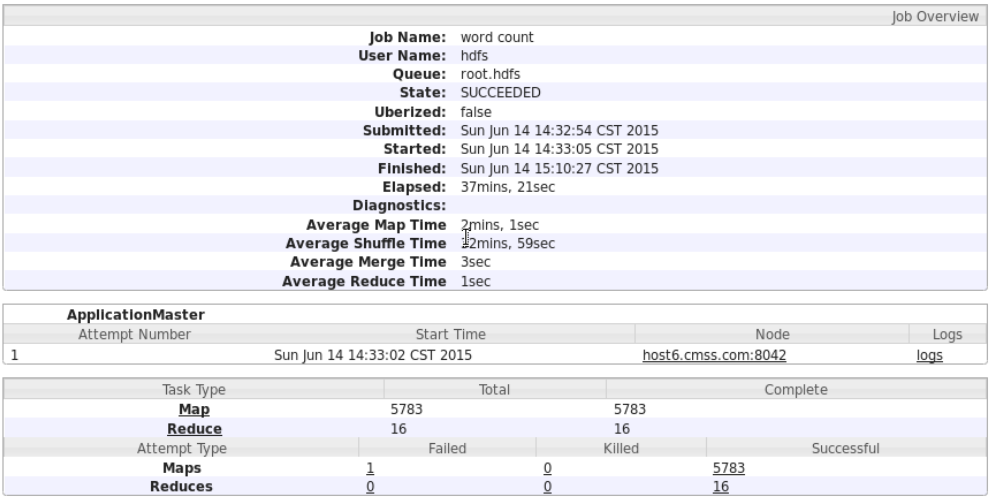
案例分析：Wordcount(文件数在 4500 左右)

参数	值
数据量	2.8T

执行参数：

参数	值
mapreduce.map.cpu.vcores	2
mapreduce.reduce.cpu.vcores	4
mapreduce.map.memory.mb	2048M
mapreduce.reduce.memory.mb	4096M
mapreduce.input.fileinputformat.split.minsize	512M
mapreduce.job.map.output.collector.class	org.apache.hadoop.mapred.nativetask.NativeMapOutputCollectorDelegator
mapreduce.task.io.sort.mb	默认值 100M，设置成 512mb
mapred.child.java.opts	1024M, Opts=sort.mb/0.8*2
mapred.reduce.tasks	16
mapreduce.map.output.compress	true
mapreduce.output.fileoutputformat.compress.codec	org.apache.hadoop.io.compress.SnappyCodec
mapreduce.reduce.shuffle.parallelcopies	50 (默认值是 30)
dfs.domain.socket.path	/var/lib/hadoop-hdfs/dn_socket

执行结果：



Job Overview

Job Name:	word count
User Name:	hdfs
Queue:	root.hdfs
State:	SUCCEEDED
Uberized:	false
Submitted:	Sun Jun 14 14:32:54 CST 2015
Started:	Sun Jun 14 14:33:05 CST 2015
Finished:	Sun Jun 14 15:10:27 CST 2015
Elapsed:	37mins, 21sec
Diagnostics:	
Average Map Time	2mins, 1sec
Average Shuffle Time	12mins, 59sec
Average Merge Time	3sec
Average Reduce Time	1sec

ApplicationMaster

Attempt Number	Start Time	Node	Logs
1	Sun Jun 14 14:33:02 CST 2015	host6.cmss.com:8042	logs

Task Type	Total	Complete
Map	5783	5783
Reduce	16	16

Attempt Type	Failed	Killed	Successful
Maps	1	0	5783
Reduces	0	0	16

问题：

Map 执行时间过长

Shuffle Time 过长（Reduce 设置过多，从结果中看 Reduce 的执行时间很短，Reduce 的处理数据量太小，Map output :500M 左右）

增加 Map 数，减少 Reduce 数目（有点冲突，需要进一步的测试）

2）Reduce 数量，一般正确的 reduce 任务个数应该是 0.95 或者 1.75*（节点数*mapred.task.tasktracker.reduce.tasks.maximum），保证 reduce 可以在一轮或者二轮内执行完。

增加 Reduce 数量会增加系统资源开销，但是可以改善负载均衡，降低任务失败带来的负面影响。

Reduce 处理的数据量介于 1-10GB

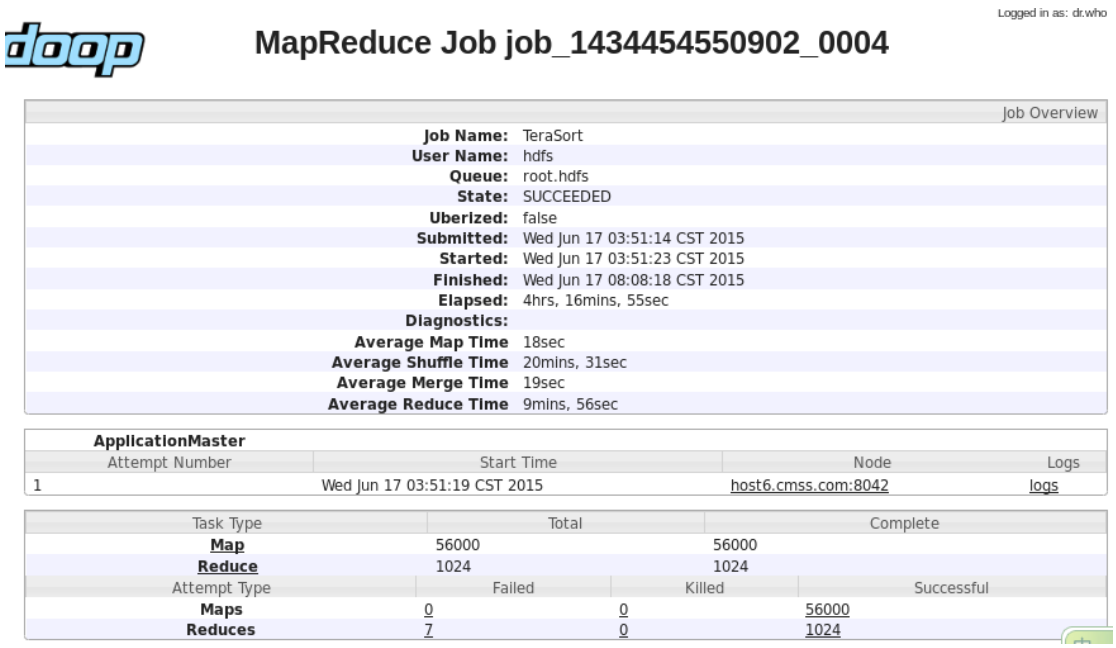
案例分析：TeraSort

参数	值
数据量	13.6TB

执行参数：

参数	值
mapreduce.map.cpu.vcores	2
mapreduce.reduce.cpu.vcores	4
mapreduce.map.memory.mb	2048M
mapreduce.reduce.memory.mb	4096M
mapreduce.input.fileinputformat.split.minsize	256M
io.sort.mb	512M
mapred.reduce.tasks	1024

执行结果：



Logged in as: dr.who

MapReduce Job job_1434454550902_0004

Job Overview	
Job Name:	TeraSort
User Name:	hdfs
Queue:	root.hdfs
State:	SUCCEEDED
Uberized:	false
Submitted:	Wed Jun 17 03:51:14 CST 2015
Started:	Wed Jun 17 03:51:23 CST 2015
Finished:	Wed Jun 17 08:08:18 CST 2015
Elapsed:	4hrs, 16mins, 55sec
Diagnostics:	
Average Map Time	18sec
Average Shuffle Time	20mins, 31sec
Average Merge Time	19sec
Average Reduce Time	9mins, 56sec

ApplicationMaster

Attempt Number	Start Time	Node	Logs
1	Wed Jun 17 03:51:19 CST 2015	host6.cmss.com:8042	logs

Task Type	Total	Complete
Map	56000	56000
Reduce	1024	1024

Attempt Type	Failed	Killed	Successful
Maps	0	0	56000
Reduces	7	0	1024

问题：

Map 执行时间太短

Shuffle 时间过长

Reduce 执行时间过长

解决：

减少 Map 数量，增加 Reduce 数量，调整参数 mapreduce.input.fileinputformat.split.minsize 及 mapred.reduce.tasks

Shuffle Time 的调整要进一步的测试

1.4 Reduce 启动时机

Reduce 需要从多个 Mapper 端 fetch 数据，在 fetch 阶段 reducetask 会启动多个 fetch 线程从所有的 mapper 端取这个 reducer 的数据，同时还有一个 fetchEvent 线程负责获取 mapper 完成的 event 通知 Fetch 线程。

Reducer 不会等到所有的 mapper 执行完毕再去拉数据，而是在 mapper task 完成一定比例后，就会开始 fetch。

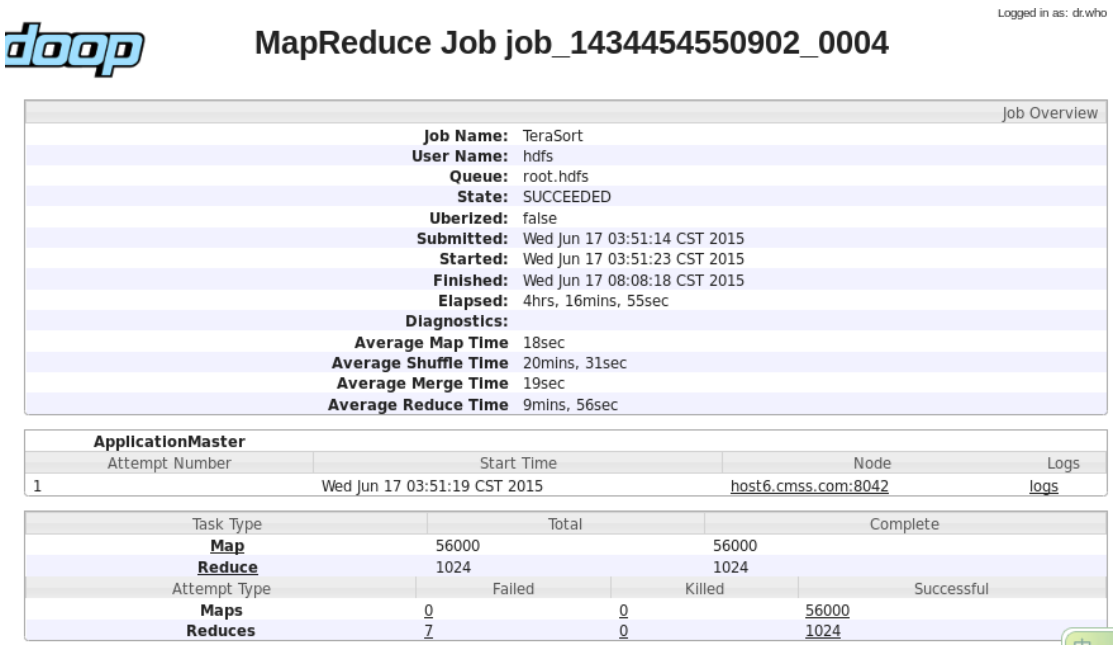
mapreduce.job.reduce.slowstart.completedmaps：当 map task 完成比例达到该值后，才会为 Reduce Task 分配资源，默认是 0.05。配置 1.0 时，reduce 要等所有的 map 完成后才开始执行，0.0 表示 reduce 从一开始就执行。

mapreduce.reduce.shuffle.parallelcopies：同时创建的 fetch 线程个数

集群网络是瓶颈时，reducer shuffle 提前，这样可以减少网络 IO 的压力，（Shuffle 阶段很长）

如果 Reduce shuffle 阶段等到 mapper finish 的时间较长，影响整个集群的运行，Reduce 占用的资源不能分配给 Map 使用

案例分析：Terasort



Logged in as: dr.who

MapReduce Job job_1434454550902_0004

Job Overview

Job Name:	TeraSort
User Name:	hdfs
Queue:	root.hdfs
State:	SUCCEEDED
Uberized:	false
Submitted:	Wed Jun 17 03:51:14 CST 2015
Started:	Wed Jun 17 03:51:23 CST 2015
Finished:	Wed Jun 17 08:08:18 CST 2015
Elapsed:	4hrs, 16mins, 55sec
Diagnostics:	
Average Map Time	18sec
Average Shuffle Time	20mins, 31sec
Average Merge Time	19sec
Average Reduce Time	9mins, 56sec

ApplicationMaster

Attempt Number	Start Time	Node	Logs
1	Wed Jun 17 03:51:19 CST 2015	host6.cmss.com:8042	logs

Task Type	Total	Complete
Map	56000	56000
Reduce	1024	1024

Attempt Type	Failed	Killed	Successful
Maps	0	0	56000
Reduces	7	0	1024

Reduce 任务相对较多，slowstart 配置较小（shuffle 开始较早），部分 Reduce 启动较早，占用较多资源。减少了 Map 运行的并行度，Map 执行较慢，已启动的 Reduce 处于等待状态，当 Reduce 空闲时间超过 mapreduce.task.timeout 配置的时间，可能超过 Reduce 被 kill 掉的情况。

因此这种情况下 slowstart.completedmaps 的配置要尽可能大，拖后 Reduce 的开始时间。在工信部的测试中，采用 0.9。

http://stackoverflow.com/questions/11672676/when-do-reduce-tasks-start-in-hadoop

2、测试中遇到的问题

2.1 datanode 处理能力

问题：

ERROR [regionserver/host15.cmss.com/192.168.1.115:60020] regionserver.HRegionServer: Shutdown / close of WAL failed:
org.apache.hadoop.hdfs.server.namenode.LeaseExpiredException:
No lease on /apps/hbase/data/oldWALs/host15.cmss.com%2C60020%2C1434252088532.default.1434252099993 (inode 155978):File is not open for writing. Holder DFSClient_NONMAPREDUCE_-25724127_1 does not have any open files

分析：

HBase Write 高并发写时，写请求处理有延迟，造成 Lease 失效

解决方法：

提高 Datanode 的处理能力，配置 dfs.datanode.max.transfer.threads（默认 1024），增加 datanode 写线程数，将该参数配置为 4096

2.2 PageRank 算法不收敛

问题：

PageRank 算法计算，经过多次迭代后，没有任何收敛

分析：

PageRank 算法的执行要经过多轮迭代，直到收敛，每轮迭代分为两个 stage，stage1 的输出是 stage2 的输入。

MR 执行是，Map 输出使用了压缩算法，mapreduce.map.output.compress=true 及 mapreduce.map.output.compress.codec ， mapreduce.output.fileoutputformat.compress.codec 为 SnappyCodec。压缩后，数据变化，导致计算出了问题。

解决方法：

去掉压缩算法，设置 mapreduce.map.output.compress 及 mapreduce.output.fileoutputformat .compress 为 false，不使用压缩算法，执行成功。

2.3 RM 自动切换

问题：

在运行 TeraSort 时，Active RM Connection 异常，StandBy RM 被激活，同时引起原执行的 Reduce 执行结果失效（Reduce 重复执行）

分析：

初步分析 Active RM 所在节点负载过程，资源被完全占满（特别是 CPU），Task 不能建立与 RM 的连接

解决方法：

降低 Active RM 所在节点的资源负载：减少可使用的 cpu 及内存（工信部测试时关掉了其所在节点的 NM）

2.4 DataNode 写入数据异常，导致 Task 失败

问题：

Error: java.io.IOException: All datanodes DatanodeInfoWithStorage[192.168.1.111:50010 ,DS-d9 8f,DISK] are bad. Aborting...

导致 Task 失败

分析：

Linux 机器打开过多的文件导致

解决方法：

```
ulimit -n
```

Linux 默认的文件打开数目是 1024

修改 /etc/security/limit.conf，增加打开的文件数目，即可

2.5 内存分配不足导致 Job 失败

问题：

TeraSort 测试中，出现以下错误：

> **mapreduce.map.memory.mb** is the upper memory limit that Hadoop allows to be allocated to a mapper, in megabytes. The default is 512. If this limit is exceeded, Hadoop will kill the mapper with an error like this:
>
> Container[pid=container_1406552545451_0009_01_000002,containerID=container_234132_0001_01_000001] is running beyond physical memory limits. Current usage: 569.1 MB of 512 MB physical memory used; 970.1 MB of 1.0 GB virtual memory used. Killing container.
>
> Hadoop mapper is a java process and each Java process has its own heap memory maximum allocation settings configured via **mapred.map.child.java.opts** (or mapreduce.map.java.opts in Hadoop 2+). If the mapper process runs out of heap memory, the mapper throws a java out of memory exceptions:

分析：

Map Task 在执行过程中，对数据的排序，对内存消耗比较大，但是配置的内存太小，因此调大 Map 使用的内存，包括：

mapreduce.map.memory.mb

mapred.child.java.opts(cdh 5.4 中，使用该参数，但是 mapred.map.child.java.opts 也可使用)

2.6 Httpd FIN_WATI1

问题：

Ambari UI 在运行过程中，出现不响应的情况（Ambari web 假死问题）

分析:

通过命令：

```
netstat -nat|awk '{print awk $NF}'|sort|uniq -c|sort -n
```

发现下面的问题：

1 State

4 CLOSE_WAIT

54 LISTEN

69 SYN_RECV

102 FIN_WAIT1

146 TIME_WAIT

155 ESTABLISHED

出现大量的 TIME_WAIT 及 FIN_WAIT，Apache 服务器负载太高

解决方法：

重启 Ambari Server (算是没有解决这个问题)

2.7 MR 任务死锁（卡住）

问题：

MR 任务不执行

分析：

Map 执行完成后，Reduce 从 output 获取文件，但是如果获取不到数据，重新启动相应 Map，Reduce 已经占用所有资源，Map 无法被调用，造成相互等待的现象，MR job 陷入死循环中。

解决：

调大 mapreduce.job.reduce.slowstart.completedmaps

2.8 Reduce Task 被 kill 掉

问题：

Task Killed dued to Timeout

分析：

Reduce 的执行要等 Map 结束，需要等待。mapreduce.task.timeout 配置任务超时时间，默认是 600s，一般是足够的，但是在工信部测试过程中，数据量较大，Map 的执行较长。

解决方法：

增加 mapreduce.task.timeout

2.9 Hive 加密不支持

Hive 的加密当前测试方法：

在 HDFS 中创建加密区（Encry cope），然后将 Hive 表数据存储到加密区中，从而达到加密的目的，但是要加密整个文件。

但是 Hive-5207,Hive-6329，Hive 9614 及 Hive 9624,etc 的工作从 Hive 表及列的层面进行加密，这些 patch 处于开发或者完成状态，不在 hive 的发布版本中。当前使用的 Hive 版本是 1.0.0，社区发布的最新版本是 1.2。Hive 的存储加密需要进一步的开发和调研。

2.10 其他问题

1） Ambari UI 缺少 Yarn Vcores 的配置项（好像已解决）

2） FreeIPA 用户认证，需要进一步的调研。Centos 6.6 FreeIPA 使用的端口冲突（Httpd:443）

3） Ambari Kerberos

启动时出现异常：*Cannot resolve servers for KDC in realm "LOCAL" while getting initial credentials*

启用后,disable 会造成 Hadoop 一样配置被修改

4）Ganglia Server rrd 文件占用空间

运行三四天后，rrd 文件，达到 30G 以上，导致了系统的崩溃