# 英特尔® Hadoop 发行版

# 版本 2.2

# 开发者指南

# Contents

# 1 开始

## 1.1 概要

本**开发者指南**提供了关于英特尔® *Hadoop* 发行版的样例代码和管理指令的实施范例，便于使用者快速熟悉英特尔® *Hadoop* 发行版。

Hadoop 为海量数据处理系统提供了超越传统内存和数据库技术的解决方案。如今，它已成为创建海量数据结构的首选工具。然而，Hadoop 的社区版本由于其开源版本本身的许多缺陷，使得企业级用户不得不为解决系统一致性，安装维护，管理以及检测的难题做出许多修改补丁。这些都使 Hadoop 的企业级应用十分困难。

针对企业用户对 Hadoop 技术平台的需要，英特尔® Hadoop 发行版产品提供了一个稳定高效可 管理的 Hadoop 发行版。英特尔® Hadoop 发行版经过大量实际项目的在线使用验证，免去了企 业用户的后顾之忧。英特尔®还提供全面的产品技术支持和顾问服务，使得企业用户在系统规划、设计、实施和运行时都能得到专业及时的专业服务。

英特尔® Hadoop 发行版能为通讯，商务，医疗，制造业等不同行业不断增长的数据处理需求提供稳定高效的技术支持。
本文档主要含有以下几个部分：

- HDFS
- MapReduce
- HBase
- ZooKeeper
- Hive
- Pig
- Mahout
- Flume
- Sqoop

**表格 1. 英特尔® Hadoop 发行版的组成结构**

| 组件名称 | 功能概要 |
|---|---|
| **HDFS** | • 针对大数据的分布式文件系统。<br>它可以使用在从几台到几千台由常规服务 器组成的集群中，并提供高聚合输入输出地文件读写访问。 |
| **Map/Reduce** | 适用于大数据量处理的分布式框架。它可以将任务分布并行运行在一个服 务器集群中 |
| **HBase** | 针对结构化和非结构化数据的分布式大数据数据库。它可以提供大数据量 的高速读写操作 |
| **ZooKeeper** | 服务于分布式系统的同步软件。它可以维护系统配置、群组用户和命名等 信息 |
| **Hive** | 大数据分布式数据仓库软件。它可以将数据存放在分布式文件 |

| | |
|---|---|
| | 系统或分布 式数据库中，并提供大数据统计、查询和分析操作 |
| **Pig** | 大数据分布式数据分析语言和运行平台。它的构架确保可以将分析任务分 布并行运行，以适应海量数据的分析需求。 |
| **Mahout** | 可扩展的机器学习类库，与 Hadoop 结合后可以提供分布式数据挖掘功能 |
| **Flume** | 提供高效采集、聚合、迁移海量日志数据的连接器模块 |
| **Sqoop** | 提供高效在 Hadoop 和结构化数据源（比如关系型数据库）之间传送数据的连接器模块 |

## 1.2 前提

为了运行英特尔® Hadoop 发行版中提供的样例程序，你需要检查如下配置：

1. 本产品说明中支持你的操作系统。
2. 英特尔® Hadoop 发行版 2.2 或以上版本已成功安装。

## 1.3 如何创建和运行一个样例程序

详细的指导见如下实例。

# 2 Hadoop

## 2.1 前提

确定 Hadoop 已被正确安装，配置，并正在运行。

## 2.2 概要

Hadoop 在底层实现了 Hadoop 分布式文件系统(HDFS)，它具有高容错率和扩展性。在 HDFS 上层有 MapReduce 服务器。MapReduce 由 JobTracker 和 TaskTracker 组成。我们可以通过 MapReduce 高效的执行计算程序。

**HDFS**

作为一个针对海量数据设计的分布式文件系统，HDFS 可以使用在几个至上千个服务器上。它为文件读写提供了高速输入输出。

HDFS 有如下几个特点

- 高容错性
  HDFS 假设系统故障是常态而非异常。它提供了许多保障数据可靠性的方法。例如，当数据被输入时，它会根据自定义的复制方案被复制多次并分布到不同的服务器中。
- 高扩展性
  数据块的分布式信息保存在 NameNode 服务器中。所以，当系统容量需要扩充时，你只需要增加 NameNode 的个数，系统会自动将新的服务器加入数列中。
- 高吞吐率
  通过使用分布式计算算法，HDFS 可以均衡地将数据存取分配到每个服务器的数据复制进程中，这可以成倍提高吞吐率
- 价位低廉
  HDFS 由低成本服务器构成。

**MapReduce**

作为一个适合处理海量数据的分布式框架，MapReduce 可以并行处理任务，并将任务分配到多个集群服务器上去。

MapReduce 适合处理复杂和大量的数据，以及提供新的分析方法。

MapReduce 框架专门处理<关键字，数值>对，就是说，该框架将任务的输入和输出当做<关键字，数值>对的集合来处理，并具有多种数据类型。

例如，一个标准的 MapReduce 处理流程如下：
输入：<关键字 1，数值 1>->映射:<关键字 2，数值 2>->合并:<关键字 2，数值 2>->约减:<关键字 3，数值 3>->输出:<关键字 3，数值 3>

## 2.3 Hadoop 提供的样例

Hadoop 如今提供它自己的样例代码。这些代码在$HADOOP_HOME /usr/lib/hadoop 目录下，包括在 hadoop-examples*.jar 和 hadoop-test*.jar 的 jar 包。

- hadoop-examples.jar

| 程序名称 | 功能 |
|---|---|
| aggregatewordcount | An Aggregate based map/reduce program that counts the words in the input file |
| aggregatewordhist | An Aggregate based map/reduce program that computes the histogram of the words in the input files |
| Dbcount | An example job that count the page view counts from a database |
| Grep | A map/reduce program that counts the matches of a regex in the input |
| Join | A job that effects a join over sorted, equally partitioned datasets |
| Multifilewc | A job that counts words from several files |
| Pentomino | A map/reduce tile laying program to find solutions to pentomino problems |
| Pi | A map/reduce program that estimates Pi using monte-carlo method |
| randomtextwriter | A map/reduce program that writes 10GB of random textual data per node |
| Randomwriter | A map/reduce program that writes 10GB of random data per node |
| Secondarysort | An example defining a secondary sort to the reduce |
| Sleep | A job that sleeps at each map and reduce task |
| Sort | A map/reduce program that sorts the data written by the random writer |
| Sudoku | A sudoku solver |
| Teragen | Generate data for the terasort |
| Terasort | Run the terasort |
| Teravalidate | Checking results of terasort |
| Wordcount | A map/reduce program that counts the words in the input files |

- hadoop-test.jar

| 程序名称 | 功能 |
|---|---|
| DFSCIOTest | Distributed i/o benchmark of libhdfs |
| DistributedFSCheck | Distributed checkup of the file system consistency |
| MRReliabilityTest | A program that tests the reliability of the MR framework by injecting faults/failures |
| TestDFSIO | Distributed i/o benchmark |

| | |
|---|---|
| Dfsthroughput | measure hdfs throughput |
| Filebench | Benchmark SequenceFile (Input\|Output) Format (block, record compressed and uncompressed), Text (Input\|Output) Format (compressed and uncompressed) |
| Loadgen | Generic map/reduce load generator |
| Mapredtest | A map/reduce test check |
| Minicluster | Single process HDFS and MR cluster |
| mrbench | A map/reduce benchmark that can create many small jobs |
| nnbench | A benchmark that stresses the NameNode |
| testarrayfile | A test for flat files of binary key/value pairs |
| testbigmapoutput | A map/reduce program that works on a very big file, which can't be split, and does identity map/reduce |
| testfilesystem | A test for FileSystem read/write. |
| testipc | A test for ipc |
| testmapredsort | A map/reduce program that validates the map-reduce framework's sort |
| testrpc | A test for RPC |
| testsequencefile | A test for flat files of binary key value pairs |
| testsequencefileinputformat | A test for sequence file input format |
| testsetfile | A test for flat files of binary key/value pairs |
| testtextinputformat | A test for text input format |
| threadedmapbench | A map/reduce benchmark that compares the performance of maps with multiple spills over maps with 1 spill |

由于许多样例测试有同样的测试点，我们选择如下典型的样例来进行功能和负载检测

## 2.4 样例: TestDFSIO (Stress Testing HDFS I/O)

TestDFSIO 样例是对 HDFS 读写功能的基准测试。通过对 HDFS 负载测试来发现集群性能的瓶颈。

注意：由于该测试作为一个 MapReduce 任务进行，该集群的 MapReduce 功能需正常工作。换句话说，这个测试不能脱离 MapReduce 进行。

你可以在如下目录中找到该样例的代码：
/usr/share/doc/hadoop-1.0.3-Intel/examples/src/test/org/apache/hadoop/fs/TestDFSIO.java

### 2.4.1 样例代码

**关键类**

| 类名 | 描述 |
|---|---|
| org.apache.hadoop.fs.TestDFSIO | 分布式 i/o 基准 |
| org.apache.hadoop.fs.IOStatMapper | 读写 mapper 基本类 |
| org.apache.hadoop.fs.WriteMapper | 写 mapper 类 |

| | |
|---|---|
| org.apache.hadoop.fs.ReadMapper | 读 mapper 类 |

### 关键方法

| 方法名 | 描述 |
|---|---|
| org.apache.hadoop.fs.TestDFSIO.createControlFile() | 初始化控制文件 |
| org.apache.hadoop.fs.TestDFSIO.analyzeResult() | 在日志中生成结果 |
| org.apache.hadoop.fs.TestDFSIO.runIOTest() | 完成对于读写的 IO 任务 |

### 源代码

样例代码为 TestDFSIO.class 中的函数:

```
    //initialize the control file
    private static void createControlFile(FileSystem fs,
                                          int fileSize, // in MB
                                          int nrFiles,
                                          Configuration fsConfig
                                          ) throws IOException {
       LOG.info("creating control file: "+fileSize+" mega bytes, "+nrFiles+"
files");
        //print the log file
       fs.delete(CONTROL_DIR, true);

       for(int i=0; i < nrFiles; i++) {
         String name = getFileName(i);
         Path controlFile = new Path(CONTROL_DIR, "in_file_" + name);
         SequenceFile.Writer writer = null;
         try {
           writer = SequenceFile.createWriter(fs, fsConfig, controlFile,
                                              Text.class, LongWritable.class,
                                              CompressionType.NONE);
        //create a new writer for sequence files
           writer.append(new Text(name), new LongWritable(fileSize));
         } catch(Exception e) {
           throw new IOException(e.getLocalizedMessage());
         } finally {
        if (writer != null)
            writer.close();
          writer = null;
         }
       }
       LOG.info("created control files for: "+nrFiles+" files");
     }
   //the main function to read & write
    private static void runIOTest(
        Class<? extends Mapper<Text, LongWritable, Text, Text>> mapperClass,
```

```
        Path outputDir,
        Configuration fsConfig) throws IOException {
  JobConf job = new JobConf(fsConfig, TestDFSIO.class);
//create a new job
  FileInputFormat.setInputPaths(job, CONTROL_DIR);
  job.setInputFormat(SequenceFileInputFormat.class);

  job.setMapperClass(mapperClass);
  job.setReducerClass(AccumulatingReducer.class);
//specify the input and output format of the file
  FileOutputFormat.setOutputPath(job, outputDir);
  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(Text.class);
  job.setNumReduceTasks(1);
  JobClient.runJob(job);
}
```

### 2.4.2 使用

通过在 HDFS 上读写文件，你可以检测集群的 HDFS 性能。

#### 运行样例

1. 运行写入测试（作为接下来读取测试的输入文件）

   运行写入测试的语法如下：

   TestDFSIO.0.0.4

   Usage: hadoop jar /usr/lib/hadoop/hadoop-test.jar TestDFSIO -read | -write | -clean [-nrFiles N] [-fileSize MB] [-resFile resultFileName] [-bufferSize Bytes]

   注意：TestDFSIO 为每个文件分配一个 map 映射，它的文件和 map 任务比例为 1:1。使用分隔符来为每一个输入输出指定一个文件名。

   例如，运行一个生成 10 个 1GB 的输出文件的写入测试指令为：

   ```
   $ hadoop jar /usr/lib/hadoop/hadoop-test.jar TestDFSIO -write -nrFiles 10 -fileSize 1000
   ```

2. 运行读取测试

   运行对应于读取 10 个 1GB 文件的指令为：

   ```
   $ hadoop jar /usr/lib/hadoop/hadoop-test.jar TestDFSIO -read -nrFiles 10 -fileSize 1000
   ```

3. 清除测试数据

清除之前测试数据的指令为：

```
$ hadoop jar /usr/lib/hadoop/hadoop-test.jar TestDFSIO -clean
```

清除任务会将 HDFS 上的文件 *directory /benchmarks/TestDFSIO* 的删除

## 分析结果

如下为 TestDFSIO_results.log 中的结果，你可以从中看见执行的细节：

```
----- TestDFSIO ----- : write
Date & time: Wed Aug 15 13:14:30 CST 2012
Number of files: 10
Total MBytes processed: 10000
Throughput mb/sec: 8.553983333418874
Average IO rate mb/sec: 8.591727256774902
IO rate std deviation: 0.5862388525690905
Test exec time sec: 145.331

----- TestDFSIO ----- : read
Date & time: Wed Aug 15 13:16:12 CST 2012
Number of files: 10
Total MBytes processed: 9005
Throughput mb/sec: 99.70022635172303
Average IO rate mb/sec: 102.70344543457031
IO rate std deviation: 34.198200892820935
Test exec time sec: 36.818
```

这里，最显眼的测量值为 *Throughput mb/sec* 和 *Average IO rate mb/sec*。它们均基于每个 map 任务的文件读写大小和使用时间，可以作为一个 Hadoop 集群的性能基准。

TestDFSIO 任务中的参数 *Throughput mb/sec* 是如下定义的。它用来估计一个 Hadoop 集群的吞吐性能。序号 1 ≤ i ≤ N 标明每个 map 任务：

$$Throughput(N) = \frac{\sum_{i=0}^{N} filesize_i}{\sum_{i=0}^{N} time_i}$$

*Average IO rate mb/sec* 参数用来估计一个集群的输入输出性能，定义如下:

$$AverageIOrate(N) = \frac{\sum_{i=0}^{N} rate_i}{N} = \frac{\sum_{i=0}^{N} \frac{filesize_i}{time_i}}{N}$$

你可能感兴趣的两个衍伸度量数值为对并发的吞吐量和集群平均输入输出能力的估计。假设你设定 TestDFSIO 样例创建 1000 个文件，但是集群中仅有 200 个映射槽。这意味着

需要使用 5 次 MapReduce waves(5 * 200 = 1000)来完成所有测试数据。在这种情况下，我们仅需使用最少文件个数(这里是 1000)，并将其与吞吐量和平均 IO 率相乘。在该例中，并发吞吐量估计值为 8.529 * 200 =1718.2 mb/s，平均并发 IO 率为 102.703*200 = 20540.6mb/s 。这些结果均为理想状态下的估计，表达式中部分数字来自 TestDFSIO_results.log。

HDFS 复制因子在其中起到了很重要的作用。如果你运行两个 TestDFSIO 样例中写入测试，它们除了 HDFS 复制因子数不同其余均相同，那么你可以看到在复制因子低的集群上运行的写入操作吞吐量和平均 IO 数更高。

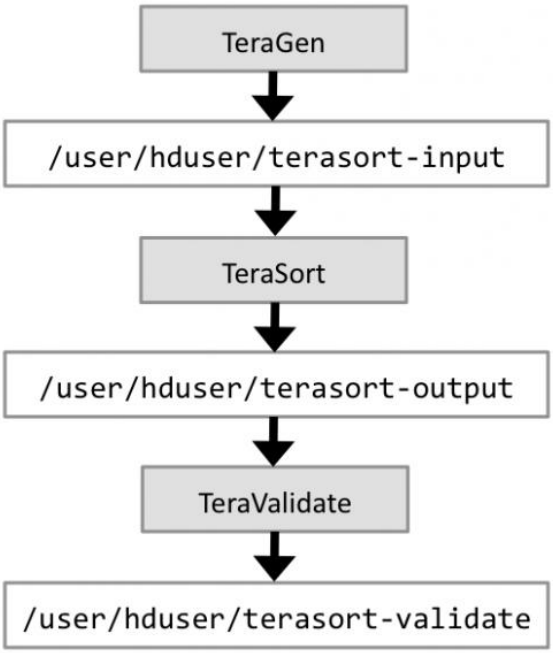
## 2.5 样例: TeraSort Benchmark Suite

TeraSort 样例的目的是将数据最快的进行排序。它是用来检测 Hadoop 集群上 HDFS 和 MapReduce 综合能力的基准测试。TeraSort 基准测试使用广泛，也得益于它还可以让我们比较不同集群的不同测试结果。TeraSort 的典型应用在帮助确定 map 和 reduce slot 分配是否合理(Map 和 reduce slot 的分配取决于每个 TaskTracker 节点所占用的内核数目和可用的 RAM 大小)，它也可以判断其他 MapReduce 相关的参数，比如 *io.sort.mb* 和 *mapred.child.java.opts* 是否设置正确。该测试也可以检测 FairScheduler 的配置是否符合预期。

一个完整的 TeraSort 基准测试有如下三步：

1. 通过 TeraGen 生成输入文件
2. 在输入文件上运行 TeraSort
3. 通过 TeraValidate 验证排好序的输出结果

下面的图片是基本的数据流程：




你可以在如下目录中找到该样例的代码：

/usr/share/doc/hadoop-1.0.3-Intel/examples/src/org/apache/hadoop/examples/terasort/TeraSort.java

### 2.5.1 样例代码

**关键类**

| 类名 | 描述 |
|---|---|
| org.apache.hadoop.examples.terasort.TeraGen | 生成正式的 terasort 输入数据集 |
| org.apache.hadoop.examples.terasort.TeraSort | 生成样例分隔点，启动任务并等待其结束 |
| org.apache.hadoop.examples.terasort.TeraValidate | 为每个文件生成一个 mapper，用来确保每个文件中关键字被正确排序 |
| org.apache.hadoop.examples.terasort.TeraInputFormat | 一种读取每行前 10 个字符作为关键字的输入格式 |
| org.apache.hadoop.examples.terasort.TeraOutputFormat | 一种流水线式文字输出格式，写入关键字，数值和"\r\n" |

**关键方法**

| 方法 | 描述 |
|---|---|
| org.apache.hadoop.examples.terasort.TeraInputFormat.writePartitionFile() | 使用输入分隔符从输入中取得样例，并生成样例关键字 |
| org.apache.hadoop.examples.terasort.TeraSort.LeafTrieNode.findPartition() | 通过字符串比较找到介于最小和最大值之间的给定关键字 |
| org.apache.hadoop.examples.terasort.TeraSort.readPartitions() | 从给定的序列文件中读取分隔点 |
| org.apache.hadoop.examples.terasort.TeraSort.buildTrie() | 提供排好序的分隔点集合，生成一个 trie，可以迅速找到正确分隔区间 |
| org.apache.hadoop.examples.terasort.TeraValidate.ValidateMapper.map() | 为每个文件生成一个 mapper，检查确保每个文件的关键字都已排好序 |
| org.apache.hadoop.examples.terasort.TeraValidate.ValidateReducer.reduce() | Reduce 方法证实所有的开始/结束项都已排好序 |

**源代码**

该样例代码在 TeraSort 类中:

```
/**
    * Given a sorted set of cut points, build a trie that will find the correct
    * partition quickly.
    * @param splits the list of cut points
    * @param lower the lower bound of partitions 0..numPartitions-1
    * @param upper the upper bound of partitions 0..numPartitions-1
    * @param prefix the prefix that we have already checked against
    * @param maxDepth the maximum depth we will build a trie for
    * @return the trie node that will divide the splits correctly
```

```
    */

private static TrieNode buildTrie(Text[] splits, int lower, int upper,
                                  Text prefix, int maxDepth) {
     int depth = prefix.getLength();
     if (depth >= maxDepth || lower == upper) {
       return new LeafTrieNode(depth, splits, lower, upper);
     }
     InnerTrieNode result = new InnerTrieNode(depth);
     Text trial = new Text(prefix);
     // append an extra byte on to the prefix
     trial.append(new byte[1], 0, 1);
     int currentBound = lower;
     for(int ch = 0; ch < 255; ++ch) {
   //partition the data to repeatedly build the trie
       trial.getBytes()[depth] = (byte) (ch + 1);
       lower = currentBound;
       while (currentBound < upper) {
         if (splits[currentBound].compareTo(trial) >= 0) {
           break;
         }
         currentBound += 1;
       }
       trial.getBytes()[depth] = (byte) ch;
       result.child[ch] = buildTrie(splits, lower, currentBound, trial,
                                    maxDepth);
     }
     // pick up the rest
     trial.getBytes()[depth] = 127;
     result.child[255] = buildTrie(splits, currentBound, upper, trial,
                                   maxDepth);
     return result;
   }
```

### 2.5.2 使用

该样例可以用来检测 Hadoop 集群中 HDFS 和 MapReduce 的性能

**运行样例**

1. **TeraGen**

TeraGen 生成随机数据，可以直接用来为接下来的 TeraSort 程序作为输入数据。

TeraGen 的运行语法如下：

```
$ hadoop jar /usr/lib/hadoop/hadoop-examples.jar teragen <number of 100-byte
rows> <output dir>
```

将 HDFS 上的输出*/user/hduser/terasort-input* 作为例子，用来运行 TeraGen 生成 1TB 输入数据的命令为：

```
$ hadoop jar /usr/lib/hadoop/hadoop-examples.jar teragen 10000000

/user/hduser/terasort-input
```

注意：TeraGen 的第一个参数为 10,000,000，不是 1GB。原因是第一个参数确定了生成的输入数据行数，每行数据大小为 100 字节。

TeraGen 每行生成的数据结构如下：

<10 bytes key> <10 bytes rowid> <78 bytes filler> \r\n

- keys 是从' '..'~'集合中生成的随机字符
- rowid 为每行的 id，格式为整数
- filler 由 7 轮从 A 到 Z 中的 10 个字符组成

启动或停止 map 任务的时间可能比实际完成任务的时间都要长。换句话说，管理 TaskTrackers 的开销可能会超过 job 的负载。一个较简单的解决方法是在 TeraGen 中增加 HDFS 块的大小。谨记 HDFS 块大小针对每一个文件，*hdfs-default.xml(*或者 *conf/hdfs-site.xml*，如果你是用的是自定义的配置文件)中的 dfs.block.size 属性值只是默认值。所以，如果你想在 TeraSort 基准测试中使用 512MB 的 HDFS 块大小(比如 536870912 bytes)，在运行 TeraGen 前重写 dfs.block.size 参数：

```
$ hadoop jar /usr/lib/hadoop/hadoop-examples.jar teragen -D

 dfs.block.size=536870912 ...
```

**2. TeraSort**

TeraSort 由一个使用 n-1 个样例 key 来定义每个 reduce 任务的 key 范围的自定义执行 MapReduce 排序 job 实现的。

运行 TeraSort 基准测试的语法如下：

```
$ hadoop jar /usr/lib/hadoop/hadoop-examples.jar terasort <input dir>
<output dir>
```

使用输入目录/user/hduser/terasort-input 和输出目录/user/hduser/terasort-output 作为样例，执行 TeraSort 的指令如下：

```
$ hadoop jar /usr/lib/hadoop/hadoop-examples.jar terasort\
```

```
/user/hduser/terasort-input /user/hduser/terasort-output
```

### 3. TeraValidate

TeraValidate 确保 TeraSort 的输出文件已被排序。

TeraValidate 在 TeraSort 输出目录中为每个文件创建一个 map 任务。一个 map 任务保证每个 key 都比原先小或者相等。Map 任务也用第一个和最后一个关键字生成记录。Reduce 任务保证文件的第一个 key 比第 i-1 个文件的最后一个 key 大。如果所有数据被正确排序，该 reduce 任务不会产生任何输出。如果检测到任何问题，它们的输出 key 是乱序的。

运行 TeraValidate 测试的语法如下：

```
$ hadoop jar /usr/lib/hadoop /hadoop-examples.jar teravalidate <terasort output dir (= input data)> <teravalidate output dir>
```

使用先前的输出目录*/user/hduser/terasort-output* 和*/user/hduser/terasort-validate* 作为例子，运行 TeraValidate 测试的命令为：

```
$ hadoop jar /usr/lib/hadoop/hadoop-examples.jar teravalidate /user/hduser/terasort-output /user/hduser/terasort-validate
```

## 分析结果

如下为 terasort-input 的部分输出：

View Next chunk

```
┆t^#\|v$2\        0AAAAAAAAAABBBBBBBBBCCCCCCCCCCDDDDDDDDDEEEEEEEEEEFFFFFFFFFFGGGGGGGGGGHHHHHHHH
75@~?'WdUF      1IIIIIIIIIIJJJJJJJJJJKKKKKKKKKKLLLLLLLLLLMMMMMMMMMNNNNNNNNNOOOOOOOOOPPPPPPPP
w[o||:N&H,      2QQQQQQQQQRRRRRRRRRSSSSSSSSSTTTTTTTTTUUUUUUUUUVVVVVVVVVWWWWWWWWWXXXXXXXX
^Eu)<n#kdP      3YYYYYYYYYZZZZZZZZZAAAAAAAAABBBBBBBBBCCCCCCCCCDDDDDDDDDEEEEEEEEEFFFFFFFF
+l-$$OE/ZH      4GGGGGGGGGHHHHHHHHHIIIIIIIIIJJJJJJJJJKKKKKKKKKLLLLLLLLLMMMMMMMMMNNNNNNNN
LsS8)|.ZLD      5OOOOOOOOOPPPPPPPPPQQQQQQQQQRRRRRRRRRSSSSSSSSSTTTTTTTTTUUUUUUUUUVVVVVVVV
1e5awB.$sm      6WWWWWWWWWXXXXXXXXXYYYYYYYYYZZZZZZZZZAAAAAAAAABBBBBBBBBCCCCCCCCCDDDDDDDD
q__[fwhKFg      7EEEEEEEEEFFFFFFFFFGGGGGGGGGHHHHHHHHHIIIIIIIIIJJJJJJJJJKKKKKKKKLLLLLLLL
;L+!2rT~hd      8MMMMMMMMMNNNNNNNNNOOOOOOOOOPPPPPPPPPQQQQQQQQQRRRRRRRRRSSSSSSSSSTTTTTTTT
M^*dDE;6^<      9UUUUUUUUUVVVVVVVVVWWWWWWWWWXXXXXXXXXYYYYYYYYYZZZZZZZZZAAAAAAAAABBBBBBBB
,C)wDw//u=     10CCCCCCCCCDDDDDDDDDEEEEEEEEEFFFFFFFFFGGGGGGGGGHHHHHHHHHIIIIIIIIIJJJJJJJJ
{570k]e<2]     11KKKKKKKKKLLLLLLLLLMMMMMMMMMNNNNNNNNNOOOOOOOOOPPPPPPPPPQQQQQQQQQRRRRRRRR
c;2;|TZ.;.     12SSSSSSSSSTTTTTTTTTUUUUUUUUUVVVVVVVVVWWWWWWWWWXXXXXXXXXYYYYYYYYYZZZZZZZZ
H.Mf,1HKf0    13AAAAAAAAAABBBBBBBBBCCCCCCCCCDDDDDDDDDEEEEEEEEEFFFFFFFFFGGGGGGGGGHHHHHHHH
q,Y!**yC_f    14IIIIIIIIIJJJJJJJJJKKKKKKKKKLLLLLLLLLMMMMMMMMMNNNNNNNNNOOOOOOOOOPPPPPPPP
=E,9!$Vl p    15QQQQQQQQQRRRRRRRRRSSSSSSSSSTTTTTTTTTUUUUUUUUUVVVVVVVVVWWWWWWWWWXXXXXXXX
_DX/C(}-4_    16YYYYYYYYYZZZZZZZZZAAAAAAAAABBBBBBBBBCCCCCCCCCDDDDDDDDDEEEEEEEEEFFFFFFFF
&F@r)QYd<x    17GGGGGGGGGHHHHHHHHHIIIIIIIIIJJJJJJJJJKKKKKKKKKLLLLLLLLLMMMMMMMMMNNNNNNNN
:x=P[~Q%D<    18OOOOOOOOOPPPPPPPPPQQQQQQQQQRRRRRRRRRSSSSSSSSSTTTTTTTTTUUUUUUUUUVVVVVVVV
>:HucdT+hR    19WWWWWWWWWXXXXXXXXXYYYYYYYYYZZZZZZZZZAAAAAAAAABBBBBBBBBCCCCCCCCCDDDDDDDD
hdk{e+Jx5L    20EEEEEEEEEFFFFFFFFFGGGGGGGGGHHHHHHHHHIIIIIIIIIJJJJJJJJJKKKKKKKKLLLLLLLL
k]-kyoX}J-    21MMMMMMMMMNNNNNNNNNOOOOOOOOOPPPPPPPPPQQQQQQQQQRRRRRRRRRSSSSSSSSSTTTTTTTT
~xs7(n7TuV    22UUUUUUUUUVVVVVVVVVWWWWWWWWWXXXXXXXXXYYYYYYYYYZZZZZZZZZAAAAAAAAABBBBBBBB
L?nNB;{*Aq    23CCCCCCCCCDDDDDDDDDEEEEEEEEEFFFFFFFFFGGGGGGGGGHHHHHHHHHIIIIIIIIIJJJJJJJJ
x`X0bkr|0A    24KKKKKKKKKLLLLLLLLLMMMMMMMMMNNNNNNNNNOOOOOOOOOPPPPPPPPPQQQQQQQQQRRRRRRRR
-#Z$UgzNNc    25SSSSSSSSSTTTTTTTTTUUUUUUUUUVVVVVVVVVWWWWWWWWWXXXXXXXXXYYYYYYYYYZZZZZZZZ
```

如下为 HDFS 上输出目录：

Go to parent directory

| Name | Type | Size | Replication | Block Size | Modification Time | Permission | Owner | Group |
|---|---|---|---|---|---|---|---|---|
| terasort-input | dir | | | | 2012-08-15 14:36 | rwxrwxrwx | root | supergroup |
| terasort-output | dir | | | | 2012-08-15 14:36 | rwxrwxrwx | root | supergroup |
| terasort-validate | dir | | | | 2012-08-15 14:39 | rwxrwxrwx | root | supergroup |

Go back to DFS home

注意：Hadoop 提供读取 job statistics 的简单命令：

$ hadoop job -history all <job output directory>

该命令可得到该 job 的历史文档(默认存储在<job output directory>/_logs/history 中的两个文件)，并且从中计算出 job statistics。

## 2.6 样例: NameNode Benchmark (nnbench)

NNBench 可有效测试 NameNode 硬件和配置的负载。它生成许多有关 HDFS 的请求，这些请求只有一个目的，就是在给 NameNode 较高的 HDFS 管理压力，从而产生较小的负载。该基准测试可以模拟创建，读取，重命名，删除文件的请求。

该测试可以同时运行在多个机器上。比如一系列的 DataNode boxes，从而同时从多个地点使用 NameNode。

你可以在如下目录中找到该样例的代码：
/usr/share/doc/hadoop-1.0.3-Intel/examples/src/test/org/apache/hadoop/hdfs/NNBench.java

### 2.6.1 样例代码

**关键类**

| 类名 | 描述 |
|---|---|
| org.apache.hadoop.hdfs.NNBench | 该程序执行一个特定的方法，可以检测 NameNode 的负载 |

**关键方法**

| 方法名称 | 描述 |
|---|---|
| org.apache.hadoop.hdfs.NNBench.createControlFiles() | 在测试运行前生成控制文件 |
| org.apache.hadoop.hdfs.NNBench.analyzeResults() | 分析输出中的结果 |
| org.apache.hadoop.hdfs.NNBench.validateInputs() | 检测输入 |
| org.apache.hadoop.hdfs.NNBench.NNBenchMapper.map() | Map方法 |
| org.apache.hadoop.hdfs.NNBench.NNBenchMapper.reduce() | Reduce方法 |

**源代码**

***Hadoop***

该样例代码在 NNBenchMapper 类中:

```
/**
    * Create and Write operation.
    * @param name of the prefix of the output file to be created
    * @param reporter an instance of (@link Reporter) to be used for
    *   status' updates
    */
   private void doCreateWriteOp(String name,
                            Reporter reporter) {
     FSDataOutputStream out;
     byte[] buffer = new byte[bytesToWrite];

     for (long l = 0l; l < numberOfFiles; l++) {
       Path filePath = new Path(new Path(baseDir, dataDirName),
              name + "_" + l);

       boolean successfulOp = false;
       while (! successfulOp && numOfExceptions < MAX_OPERATION_EXCEPTIONS) {
         try {
           // Set up timer for measuring AL (transaction #1)
           startTimeAL = System.currentTimeMillis();
           // Create the file
           // Use a buffer size of 512
           out = filesystem.create(filePath,
                  true,
                  512,
                  replFactor,
                  blkSize);
           out.write(buffer);
           totalTimeAL1 += (System.currentTimeMillis() - startTimeAL);

           // Close the file / file output stream
           // Set up timers for measuring AL (transaction #2)
           startTimeAL = System.currentTimeMillis();
           out.close();
           //store the total time for the write operation
           totalTimeAL2 += (System.currentTimeMillis() - startTimeAL);
           successfulOp = true;
           successfulFileOps ++;

           reporter.setStatus("Finish "+ l + " files");
         } catch (IOException e) {
           LOG.info("Exception recorded in op: " +
                  "Create/Write/Close");
```

```
          numOfExceptions++;
        }
      }
    }
  }
```

### 2.6.2 使用

NNBench 的使用语法如下：

```
NameNode Benchmark 0.4
Usage: $ hadoop jar /usr/lib/hadoop/hadoop-test.jar mrbench <options>
Options:
      -operation
                <Available operations are create_write open_read rename delete.
                 This option is mandatory>
       * NOTE: The open_read, rename and delete operations assume that the files
               they operate on, are already available. The create_write operation
               must be run before running the other operations.
      -maps
                <number of maps. default is 1. This is not mandatory>
      -reduces
                <number of reduces. default is 1. This is not mandatory>
      -startTime
                <time to start, given in seconds from the epoch. Make sure this
                 is far enough into the future, so all maps (operations) will
                 start at the same time>. default is launch time + 2 mins. This
                 is not mandatory
      -blockSize
                <Block size in bytes. default is 1. This is not mandatory>
      -bytesToWrite
                <Bytes to write. default is 0. This is not mandatory>
      -bytesPerChecksum
                <Bytes per checksum for the files. default is 1. This is not
                 mandatory>
      -numberOfFiles
                <number of files to create. default is 1. This is not mandatory>
      -replicationFactorPerFile
                <Replication factor for the files. default is 1. This is not
                 mandatory>
      -baseDir
                <base DFS path. default is /becnhmarks/NNBench. This is not
                 mandatory>
```

```
-readFileAfterOpen
            <true or false. if true, it reads the file and reports the average
             time to read. This is valid with the open_read operation.
             default is false. This is not mandatory>
-help: Display the help statement
```

## 运行样例

下面的命令将运行一个 NameNode 基准测试，该测试使用 12 个 map 和 5 个 reducer 生成 1000 个文件。它根据机器的简写主机名生成自定义的输出目录。这是一个确保一个 box 不会意外写入其他 box 同时运行的 NNBench 测试的同名输出目录的技巧。

```
$ hadoop jar /usr/lib/hadoop/hadoop-test.jar nnbench -operation create_write
-maps 12 -reduces 6 -blockSize 1 -bytesToWrite 0 -numberOfFiles 1000
-replicationFactorPerFile 3 -readFileAfterOpen true -baseDir
/benchmarks/NNBench- 'hostname -s'
```

注意：默认情况下 benchmark 会在运行前等待 2 分钟。

## 分析结果

Shell 中部分输出：

```
Test Inputs:
INFO hdfs.NNBench:           Test Operation: create_write
INFO hdfs.NNBench:               Start time: 2012-08-15 15:59:07,620
INFO hdfs.NNBench:          Number of maps: 12
INFO hdfs.NNBench:        Number of reduces: 6
INFO hdfs.NNBench:               Block Size: 1
INFO hdfs.NNBench:            Bytes to write: 0
INFO hdfs.NNBench:       Bytes per checksum: 1
INFO hdfs.NNBench:          Number of files: 1000
INFO hdfs.NNBench:        Replication factor: 3
INFO hdfs.NNBench:                Base dir: /benchmarks/NNBench-hbase1
```

HDFS 信息网页上的内容：

Contents of directory /benchmarks/NNBench-hbase1

Goto : /benchmarks/NNBench go

Go to parent directory

| Name | Type | Size | Replication | Block Size | Modification Time | Permission | Owner | Group |
|---|---|---|---|---|---|---|---|---|
| control | dir | | | | 2012-08-15 15:57 | rwxrwxrwx | root | supergroup |
| data | dir | | | | 2012-08-15 16:00 | rwxrwxrwx | root | supergroup |
| output | dir | | | | 2012-08-15 16:00 | rwxrwxrwx | root | supergroup |

Go back to DFS home

## 2.7 样例: MapReduce Benchmark (mrbench)

MRBench 反复多次运行一个小型 job。相比大规模的 TeraSort 基准测试，更加简单。MRBench 判断在你的集群上小型 job 是否反应迅速且运行高效。它将重点放在 MapReduce 层上，对于 HDFS 层的影响很小。

你可以在如下目录中找到该样例的代码：
/usr/share/doc/hadoop-1.0.3-Intel/examples/src/test/org/apache/hadoop/mapred/MRBench.java

### 2.7.1 样例代码

**关键类**

| 类名 | 描述 |
|---|---|
| org.apache.hadoop.mapred.MRBench | 多次运行同一个任务并取得平均值 |

**关键方法**

| 方法名称 | 描述 |
|---|---|
| org.apache.hadoop.mapred.MRBench.pad() | 将给定数字转化为字符串并用0封装在首位，使得字符串长度规整 |
| org.apache.hadoop.mapred.MRBench.setupJob() | 创建job的配置 |
| org.apache.hadoop.mapred.MRBench.runJobInSequence() | 多次运行一个MapReduce任务，每次的输入为同一个文件 |

**源代码**

如下为 MRBench.java 中的 runJobInSequence 函数：

```
/**
   * Runs a MapReduce task, given number of times. The input to each run
   * is the same file.
   */
```

```
 private ArrayList<Long> runJobInSequence(JobConf masterJobConf, int numRuns)
throws IOException {
   Random rand = new Random();
   ArrayList<Long> execTimes = new ArrayList<Long>();

   for (int i = 0; i < numRuns; i++) {
     // create a new job conf every time, reusing same object does not work
     JobConf jobConf = new JobConf(masterJobConf);
     // reset the job jar because the copy constructor doesn't
     jobConf.setJar(masterJobConf.getJar());
     // give a new random name to output of the mapred tasks
     FileOutputFormat.setOutputPath(jobConf,
                     new Path(OUTPUT_DIR, "output_" + rand.nextInt()));

     LOG.info("Running job " + i + ":" +
             " input=" + FileInputFormat.getInputPaths(jobConf)[0] +
             " output=" + FileOutputFormat.getOutputPath(jobConf));

     // run the mapred task now
     long curTime = System.currentTimeMillis();
     JobClient.runJob(jobConf);
     execTimes.add(new Long(System.currentTimeMillis() - curTime));
   }
   return execTimes;
 }
```

### 2.7.2 使用

该样例可在单个 box 上运行(见下列注意事项)。命令语法可在 mrbench -help 中列出：

```
MRBenchmark.0.0.2
Usage: mrbench
       [-baseDir ]
       [-jar ]
       [-numRuns ]
       [-maps ]
       [-reduces ]
       [-inputLines ]
       [-inputType ]
       [-verbose]
```

重要注释：在 Hadoop0.20.*中，设置-baseDir 参数没有任何作用。这意味着多个平行 MRBench 运行(例如从多个不同 boxes 开始)可能会互相影响。这是一个已知的 bug (MAPREDUCE-2398)。

默认参数值为：

-baseDir: /benchmarks/MRBench
-numRuns: 1
-maps: 2
-reduces: 1
-inputLines: 1
-inputType: ascending

### 运行样例

运行连续 50 个小测试 job 的命令为：

```
$ hadoop jar /usr/lib/hadoop/hadoop-test.jar mrbench -numRuns 50
```

### 分析结果

结果如下：

```
DataLines       Maps    Reduces    AvgTime (milliseconds)
1                2        1          25254

This means that the average finish time of executed jobs was 25254 seconds.
```

## 2.8 样例: MapReduce Jar Addition

当你运行一个 MapReduce 任务时，有时会需要第三方 jar 包。本样例提供两种方法解决在 MapReduce 任务中使用第三方 jar 包的问题。

### 2.8.1 样例代码

**关键类**

| 类名 | 描述 |
|---|---|
| wordCount | 计算输入文件中不同单词的个数 |
| ToLowCase | 将单词转化为小写 |

**关键方法**

| 方法名称 | 描述 |
|---|---|
| wordCount::map() | The mapper |
| wordCount::reduce() | The reducer |

### 源代码

源代码分成了如下 3 个部分:

*wordCount.class*

*myHadoopJob.class*

*ToLowCase.class*

如下为方法一中的 wordCount.class

```
import java.io.IOException;
import java.util.*;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import Test.MapReduce.*;    //import the third-party jar

public class WordCount {
```

```
  public static class Map extends MapReduceBase implements
      Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value,
        OutputCollector<Text, IntWritable> output, Reporter reporter)
        throws IOException {

      String line = value.toString();
      line = ToLowCase.run(line);//call the function in the third-party jar

      StringTokenizer tokenizer = new StringTokenizer(line);
      while (tokenizer.hasMoreTokens()) {
        word.set(tokenizer.nextToken());
        output.collect(word, one);
      }
    }
  }

  public static class Reduce extends MapReduceBase implements
      Reducer<Text, IntWritable, Text, IntWritable> {
    public void reduce(Text key, Iterator<IntWritable> values,
        OutputCollector<Text, IntWritable> output, Reporter reporter)
        throws IOException {
      int sum = 0;
      while (values.hasNext()) {
        sum += values.next().get();
      }
      output.collect(key, new IntWritable(sum));
    }
```

```
  }
  public static void main(String[] args) throws Exception {
    JobConf conf = new JobConf(WordCount.class);
    conf.setJobName("wordcount");

    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);

    conf.setMapperClass(Map.class);
    conf.setCombinerClass(Reduce.class);
    conf.setReducerClass(Reduce.class);
```

```
    conf.setInputFormat(TextInputFormat.class);
    conf.setOutputFormat(TextOutputFormat.class);

    FileInputFormat.setInputPaths(conf, new Path(args[0]));
    FileOutputFormat.setOutputPath(conf, new Path(args[1]));

    JobClient.runJob(conf);
  }
}
```

如下为方法二中 *myHadoopJob.class*:

```
import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
```

```
import Test.MapReduce.ToLowCase;

public class MyHadoopJob extends Configured implements Tool {

  public static class MapClass extends
      Mapper<LongWritable, Text, Text, IntWritable> {

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {
      String line = value.toString();

      line = ToLowCase.run(line);

      StringTokenizer tokenizer = new StringTokenizer(line);
      while (tokenizer.hasMoreTokens()) {
        word.set(tokenizer.nextToken());
        context.write(word, one);
      }
    }
  }

  public static class ReduceClass extends
      Reducer<Text, IntWritable, Text, IntWritable> {

    public void reduce(Text key, Iterable<IntWritable> values,
        Context context) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      context.write(key, new IntWritable(sum));
    }
  }

  @Override
  public int run(String[] args) throws Exception {
    System.out.println(args.length);
    Configuration conf = getConf();
    Job job = new Job(conf, "MyHadoopJob");
```

```
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    job.setJarByClass(MapClass.class);
    job.setJarByClass(ReduceClass.class);

    job.setMapperClass(MapClass.class);
    job.setReducerClass(ReduceClass.class);

    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);

    FileInputFormat.addInputPath(job, new Path("/user/input"));
    FileOutputFormat.setOutputPath(job, new Path("/user/output"));

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

  public static void main(String args[]) throws Exception {
    int res = ToolRunner.run(new Configuration(), new MyHadoopJob(), args);
    System.exit(res);
  }
}
```

如下为第三方 jar 包中 ToLowCase.class 代码：

```
package Test.MapReduce;
//convert the words to lower case
public class ToLowCase {
  public static String run(String s){
        return s.toLowerCase();
  }
}
```

### 2.8.2 使用

该样例提供了两种方法在 MapReduce 任务中调用第三方 jar 包。

#### 运行样例

**方法一**

1. 在编写 mapreduce 任务时，在工程下新建 lib 文件夹，并将所需第三方 jar 包复制在 lib 文件夹下，然后将这些 jar 包添加到工程中。

在样例中，当你导入所有需要的 jar 包后，你可以将整个任务生成一个新的最终 jar 包，然后就可以在集群上运行 mapreduce 任务了。(本例中生成 jar 包为 mp_lib.jar，若你需要详细代码，见源代码第一三部分)

2. 创建两个文件作为输入文件，并将其复制到 HDFS 中。例如，你可以拷贝本地文件 file01 和 file02 到 HDFS 的目录/user/input 下：

```
hadoop fs -copyFromLocal input/file01 /user/input
hadoop fs -copyFromLocal input/file02 /user/input
```

3. 你可以用如下指令查看文件内容，如下为本例中显示：

```
[root@hadoop ~]# hadoop fs -cat /user/input/file01
hello world hadoop hi

[root@hadoop ~]# hadoop fs -cat /user/input/file02
Hello world hadoopall
```

4. 现在你可以运行 MapReduce 任务。在本样例中，该任务可以将输入文件中的单词转化为小写并统计不同单词的出现次数：

```
hadoop fs -rmr /user/output
hadoop jar mp_lib.jar WordCount /user/input/ /user/output
```

5. 通过如下指令查看结果：

```
hadoop fs -cat /user/output/part-00000
```

**方法二**

1. 如果当第三方 jar 包特别巨大，或者不在本地传输缓慢，导致开发进程受阻的时候，就不应该再将第三方 jar 包打入自己的程序中，而应该采用其他的办法。
   Hadoop 已经提供了相应的接口，具体办法为以下几步：

   - mapreduce 主程序必须是在 main 方法中使用以下的方法来调用：

     ```
     ToolRunner.Run(conf, tool, args);
     ```

     其中，conf 为配置 Configuration() ；tool 为主程序的 new 实例，该主程序必须包含 run(String args[])方法；args 为带入的参数，可以代入 mapreduce 的输入输出路径等。

     必须这么做的原因是，-libjars path 这种方式，必须通过 ToolRunner.Run() 这个方法的调用才能生效，具体的，是在这其中的 new GenericOptionsParser(conf, args);方法里得到了 jar 包的路径设置。

   - 实现相应的 map 和 reduce 类，此处一定要注意一个很关键的问题。

     ```
     Job job = new Job(getConf(), "MyHadoopJob");
     job.setJarByClass(MapClass.class);
     job.setJarByClass(ReduceClass.class);
     job.setMapperClass(MapClass.class);
     ```

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

当采用这种方式来把 mapreduce 任务加载进 job 时，一定要注意，mapClass 类必须是 org.apache.hadoop.mapreduce.Mapper 的继承类，reduceClass 必须是 org.apache.hadoop.mapreduce.Reducer 的继承类。

注意：不要使用 org.apache.hadoop.mapred.Mapper 这个接口并继承 org.apache.hadoop.mapred.MapReduceBase 来实现 map 类，如果这样，是无法加载到 Job 这个类里面去的。Reduce 类也是类似的。

2. 现在你可以运行 MapReduce 任务(你需要重复方法一中的步骤 2 和 3)。在本样例中，MapReduce 任务可以将输入文件中的单词转化为小写并统计不同单词的出现次数。(若你需要详细代码，见源代码）

```
hadoop fs -rmr /user/output
hadoop jar MR.jar MyHadoopJob -libjars tolowcase.jar
```

3. 通过如下指令查看结果：

```
hadoop fs -cat /user/output/part-00000
```

### 分析结果

在没有调用第三方 jar 包的运行结果：

```
Hello        1
hadoop       1
hadoopall    1
hello        1
hi           1
world        2
```

调用第三方 jar 包的运行结果：

```
hadoop        1
hadoopall     1
hello         2
hi            1
world         2
```

## 2.9 样例:Key Management

根据对 HDFS 加密技术的讨论以及对 Key Management 的回顾，我们将在应用层使用 CompressCodec 这个框架来完成数据的加密和解密。

我们必须在各层对 Key Management 提供简单、灵活的方法。

1) HDFS 通过文件系统的 API 的连接

2) HDFS 通过 Map Reduce 的连接

- 当我们直接通过文件系统的 API 来连接 HDFS 的时候，数据在客户端被加密和解密

名词解释：

- Key：该类储存了用于加密算法的密钥，这些密钥有如下属性：
  - Cryptogrphic Algorithm：使用的 alogirhtm，例如 DES, 3DES, or AES
  - Cryptograhphic Length：密钥的长度，例如 128, 192, 256
  - Key Data：密钥的原始数据
- KeyProfile：key profile 是一些比如与加密数据相关联并储存的密钥 ID 的一些数据，一个 KeyProfile 并不是加密数据时所采用的密钥，但它与密钥建立了某种关联。
- KeyProfileResolver：它提供了一种回调机制，用于解决使用某个密钥储存的 KeyProfile，然后可以用于解密数据。

- 在 Map Reduce 时加密和解密文件，我们有以下要求：
  - 在不同的阶段（例如 input, output, intermediate output），应该能够灵活的选择加密与否。
  - 不同的文件（例如不同的输入文件），应该拥有并使用不同的 key。

### 2.9.1 样例代码

我们采用一个简单的例子来阐述如何使用 Key Manager 来加密与解密一个文件。

本例是将"hello, encryption \n" 这个字符串通过密码"12345678"来加密保存至 hdfs 文件系统，再将其解密。

**源代码**

```
public class EncryptionTest {
  //设置解密与解密的地址
  static Path encryptionPath = new Path("/user/test/Encryption");
  static Path decryptionPath = new Path("/user/test/Decryption");

  static AESCodec aesCodec ;
  static FileSystem fs;
  //初始化环境
  private static void init() throws IOException{
    Configuration conf = new Configuration();
    //第二个参数为namenode上hdfs的端口
    conf.set("fs.default.name", "hdfs://appleminy.sh.intel.com:8020");
    fs = FileSystem.get(conf);

    aesCodec = new AESCodec();
    CryptoContext cryptoContext = new CryptoContext();
    Key key;
    try {
      //设置密钥
```

```
      key = Key.derive("12345678");
    } catch (CryptoException e) {
      throw new IOException(e);
    }
    cryptoContext.setKey(key);
    aesCodec.setCryptoContext(cryptoContext);
  }

  public static void testEncryption()
      throws IOException {
    FSDataOutputStream out = fs.create(encryptionPath, true);
    CompressionOutputStream compressionOut = aesCodec
        .createOutputStream(out);
    //将要加密保存的内容
    String hello = "hello, encryption \n";
    compressionOut.write(hello.getBytes("utf-8"));
    compressionOut.close();
    fs.close();
  }
  public static void testDecryption() throws IOException{
    byte[] buffer = new byte[1024 * 1024]; //64k
    InputStream inStream = aesCodec.createInputStream(fs.open(encryptionPath));
    OutputStream outStream = fs.create(decryptionPath, true);
    int read = 0;
       while(0 < (read = inStream.read(buffer, 0, 64 * 1024))) {
           outStream.write(buffer, 0, read);
       }
       outStream.flush();
       outStream.close();
  }
  public static void main(String[] args) {
    System.out.print("Usage: \n" +
             "Encrypt: {this binary} -e \n" +
         "Decrypt: {this binary} -d \n");
    boolean encrypt = args[0].equals("-e");
    boolean decrypt = args[0].equals("-d");
    try {
      init();
      if (encrypt){testEncryption();}
      if (decrypt) {testDecryption();}
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
```

***Hadoop***

```
}
```

### 2.9.2 使用

1) 将上述代码写入工程，关联/usr/lib/hadoop 以及/usr/lib/hadoop/lib 下的以下这些 jar 包文件：

   hadoop-core*.jar

   commons-configuration*.jar

   commons-lang*.jar

   commons-longging*.jar

   guava*.jar

2) 运行工程中 EncryptionTest 这个类，设置 VM arguments：

   ```
   -Djava.library.path=/usr/lib/hadoop/lib/native/Linux-amd64-64/
   ```

3) 首先，第一次运行，选择加密，运行参数为：

   -e

   运行结果，在命令行输入：

   hadoop fs -cat /user/test/Encryption

   终端输出结果：

   我们可以看到，保存在 hdfs 上的是经过加密的文件

4) 然后，第二次运行，选择解密，运行参数为：

   -d

   运行结果，在命令行输入：

   hadoop fs -cat /user/test/Decryption

   终端输出结果：

   ```
   [caijie@appleminy Desktop]$ hadoop fs -cat /user/test/Decryption
   hello, encryption
   ```

   我们可以看到，解密成功。

# 3 HBase

## 3.1 前提

确定 Hadoop，Hive，HBase 已经被正确安装，配置，并且正在运行。推荐使用有三个虚拟机的集群。

## 3.2 概要

HBase 是一个面向列的分布式数据库。 HBase 不是一个关系型数据库，其设计目标是用来解决关系型数据库在处理海量数据时的理论和实现上的局限性。HBase 从一开始就是为 Terabyte 到 Petabyte 级别的海量数据存储和高速读写而设计，这些数据要求能够被分布在数千台普通服务器上，并且能够被大量并发用户高速访问。

特别地，HBase 是一个面向列的、稀疏的、分布式的、持久化存储的多维排序映射表（Map）。表的索引是行关键字、列簇名（Column Family）、列关键字以及时间戳；表中的每个值都是一个未经解析的字节数组。

HBase 是由 Hmaster 和 Hregionserver 组成。Hmaster 负责给 region server 分配 region，并且处理对于架构变动的元数据操作，以及表单和列簇的创建。通过处理对于所有 region 的读写请求，Hmaster 也负责处理 region server 上 region 的负载均衡。一旦 region 的大小超过了阈值，Hregionserver 会将 region 分裂。

## 3.3 输入输出

HBase 可以简易的展示表格结果。Table 中的所有行都按照 row key 的字典序排列。

Table 在行的方向上分割为多个 Hregion。region 按大小分割的，每个表一开始只有一个 region，随着数据不断插入表，region 不断增大，当增大到一个阀值的时候，Hregion 就会等分会两个新的 Hregion。当 table 中的行不断增多，就会有越来越多的 Hregion。

## 3.4 样例: HBase CreateTable

HBase CreateTable 样例通过在你的集群中创建一张新的表来测试 HBase 的基本功能。此样例也可演示英特尔 Hadoop 发行版特有的跨数据中心大表创建功能。但如果你需要跨数据中心大表功能，你至少需要两个 HBase 集群，并按照下文“运行样例”步骤 4 完成启用跨数据中心功能。

你可以在随本文档一起提供的 examples.tar.gz 中找到此该样例的可运行代码。

### 3.4.1 样例代码

**关键类**

| 类名 | 描述 |
|---|---|
| com.intel.hbase.table.create.Configure | 设置程序的基础配置 |
| com.intel.hbase.table.create.TestCreateTable | 在 HDFS 中创建表格 |
| com.intel.idh.test.util.ResultHTMLGenerater | 将将结果输出到 HTML 文件中 |

**关键方法**

| 方法名称 | 描述 |
|---|---|
| com.intel.hbase.table.create.TestCreateTable::createTable() | 创建一个新的表格 |
| com.intel.hbase.table.create.TestCreateTable::genHTableDescriptor() | 得到新表格的描述符 |
| com.intel.hbase.table.create.TestCreateTable::existsFamilyName() | 判断该列簇名是否存在 |
| com.intel.idh.test.util.ResultHTMLGenerater::generateHTMLFile() | 创建 HTML 文件的框架 |
| com.intel.idh.test.util.ResultHTMLGenerater::printInfo() | 将表格的内容加入 HTML 文件中 |

**源代码**

下面两个方法在 *com.intel.hbase.table.create.TestCreateTable* 类中：

```
//generate the descriptor of the table
public static HTableDescriptor genHTableDescriptor(String tableName) {

//create the table
    HTableDescriptor ht = new HTableDescriptor(tableName);
    //create the column family of the table
    HColumnDescriptor desc = new HColumnDescriptor(Configure.FAMILY_NAME);
    Configure.configColumnFamily(desc);
    ht.addFamily(desc);
    return ht;
  }
//use the descriptor to create the table
public static void createTable(String tableName) {

    boolean result = false;

    try {
      removeTable(tableName);
      hba.createTable(genHTableDescriptor(tableName));
    //check if the table is created successfully
      result = hba.tableExists(tableName);
    } catch (IOException e) {
```

```
      e.printStackTrace();
    } finally {
      rg.addCase("CreateTable",
          "Create Table using createTable(HTableDescriptor desc) ",
          result);
    }
  }
public static boolean existsFamilyName(HBaseAdmin hba, String tableName,
      String columnName) {
    HTableDescriptor[] list;
    try {
      list = hba.listTables();
      for (int i = 0; i < list.length; i++) {
        if (list[i].getNameAsString().equals(tableName))
          for (HColumnDescriptor hc : list[i].getColumnFamilies()) {
            if (hc.getNameAsString().equals(columnName))
              return true;
          }
      }
    } catch (IOException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    }
    return false;
  }

public void generateHTMLFile(String file) {
    StringBuffer sb = new StringBuffer();
    sb.append("<html>");
    sb.append("<head>");
    sb.append("<title>" + title + "</title>");
    sb.append("</head>");
    sb.append("<body>");
    sb.append("<table border=1>");
    for (ResultInfo info: this.info ) {
      printInfo(sb, info);
    }
    sb.append("</table>");
    sb.append("</body>");
    try {
      FileWriter fw = new FileWriter(file);
      fw.append(sb);
      fw.close();
    } catch (IOException e) {
```

```
    e.printStackTrace();
  }
}



private void printInfo(StringBuffer sb, ResultInfo info) {
  sb.append("<tr>");
  sb.append("<td>" + info.name + "</td>");
  sb.append("<td>" + info.desciption + "</td>");
  if (info.pass) {
    printPassed(sb);
  } else {
    printFailed(sb);
  }
  sb.append("</tr>");
}
```

### 3.4.2 使用

该样例用来创建一个新的表格并将其展示在 HTML 网页中。

#### 运行样例

1) 将包含样例代码的文件导入你已经装好英特尔® Hadoop 发行版 (IDH)的集群中。

2) 将在目录*/etc/hbase/conf/*下的文件 *hbase-site.xml* 拷贝到目录 *conf/* 下。

3) 请确保样例代码的 lib 目录下的所有 jar 包与集群 jar 包保持一致，如果已经运行过本样例则可跳过这个步骤。

   检查*/root/usr/lib/*目录下的 jar 包，比如 hbase_VERSION.jar 和 zookeeper_VERSION.jar，若是它们的版本号和样例程序中的不同，你需要将不同的 jar 包拷贝到样例代码的 lib 目录下并且更改 build.xml 文件中的 jar 包名称。

4) 打开浏览器，登陆页面： *https://host:9443*. 进入目录 *Configuration/hbase/Full Configuration* 并且将 *hbase.use.partition.table* 设置为 *true*。你需要将与你集群关联的 ZooKeeper 地址填入 *hbase.partition.zookeeper* 中。保存你的更改并点击 configure the cluster。

   注意：你也许需要重启你的 HBase 服务来使更改生效。你可以在 shell 中使用命令 service hbase-master restart。

5) 运行指令:

   ```
   ant TestCreateTable
   ```

6) 新的表格可以在当前目录下的文件 result.html 中找到。

### 分析结果

页面 result.html 结果如下:

| | | |
|---|---|---|
| CreateTable | Create Table using createTable(HTableDescriptor desc) | PASSED |
| CreateTableWithSplitKeys | Create Table using createTable(HTableDescriptor desc, byte [][] splitKeys):Test_Table_SpilitKey | PASSED |
| CreateTableWithStartAndEndKey | Create Table using createTable(HTableDescriptor desc, byte[] startKey, byte[] endKey, int numRegions):Test_Table_StartKey_EndKey_Num | PASSED |
| add column test1 | add column test1 to Test_Table get exception | PASSED |
| delete column test1 | delete column test1 from Test_Table get exception | PASSED |
| disable table | disable table Test_Table | PASSED |
| add column test2 | add column test2 to Test_Table | PASSED |
| delete column test2 | delete column test1 from Test_Table | PASSED |
| enable table | enable table Test_Table | PASSED |

你可以在 *result.html* 页面中检查是否所有的指令均已运行成功。.

## 3.5 样例: HBase Replication

HBase Replication sample 通过在集群中复制一个已存在的表格测试 HBase 的基本功能。你可以更改样例中参数来改变复制次数。

你可以在随本文档一起提供的 examples.tar.gz 中找到此该样例的可运行代码。

### 3.5.1 样例代码

**关键类**

| 类名 | 描述 |
|---|---|
| com.intel.hbase.table.create.Configure | 设置程序的基本配置 |
| com.intel.hbase.table.replication.TestReplication | 测试 HDFS 的 replication 功能 |

**关键方法**

| 方法名称 | 描述 |
|---|---|
| com.intel.hbase.table.create.Configure:: genHTableDescriptor() | 得到新表格的描述符 |
| com.intel.hbase.table.create.Configure::getHBaseConfig() | 返回目前配置的_config 参数 |

**源代码**

下面的方法在 *com.intel.hbase.table.create.TestReplication* 类中.

```
public class TestReplication {
  private static String tableName = "replication_1";
```

```
  public static void main(String[] agrs) {
    HBaseAdmin hba = null;
    HTable table = null;
    try {
      hba = new HBaseAdmin(Configure.getHBaseConfig());
      //create a new HBase Admin
      hba.createTable(Configure.genHTableDescriptor(tableName,
(short)1));//replicate one new table
      table = new HTable(Configure.getHBaseConfig(), tableName);
      Put put = new Put(Bytes.toBytes("007"));//add a new record, whose rowkey
is '007'
      put.add(Bytes.toBytes(Configure.FAMILY_NAME),
Bytes.toBytes("key"),Bytes.toBytes("value"));
      table.put(put);
      table.flushCommits();
      hba.disableTable(tableName);

    } catch (MasterNotRunningException e) {
      e.printStackTrace();
    } catch (ZooKeeperConnectionException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    } finally {
      if(hba != null) {
        try {
          hba.close();//close the HBaseAdmin
        } catch (IOException e) {
        }
      }
       if (table != null) {
         try {
          table.close();//close the table
        } catch (IOException e) {
        }
       }

    }
  }
}
```

### 3.5.2 使用

通过修改复制次数的参数，你可以检测 HBase 组件复制准确度。

***HBase***

## 运行样例

1) 将包含样例代码的文件导入你已经装好英特尔® Hadoop 发行版 (IDH)的集群中。

2) 将在目录*/etc/hbase/conf/*下的文件 *hbase-site.xml* 拷贝到目录 *conf/*下

3) 请确保样例代码的 lib 目录下的所有 jar 包与集群 jar 包保持一致，如果已经运行过本样例则可跳过这个步骤。

检查*/root/usr/lib/*目录下的 jar 包，比如 hbase_VERSION.jar 和 zookeeper_VERSION.jar，若是它们的版本号和样例程序中的不同，你需要将不同的 jar 包拷贝到样例代码的 lib 目录下并且更改 build.xml 文件中的 jar 包名称。

4) 检查集群中是否已经存在表格 replication_one，若存在，请删除，否则会报错 *Table Exist Exception*。

5) 执行指令:

```
ant TestTableReplication
```

6) 打开浏览器，登陆页面： *https://host:9443* 在左侧选择：集群概况---HDFS 概述，在上侧选择 HDFS 浏览器

7) 进入目录 */hbase/*replication_one/ 来检查在 *family* 文件中的复制次数是否为 1.

8) 现在你可以修改 文件中的复制次数参数值为 0 或负值，并再次运行样例。

## 分析结果

网页结果如下：


如果你将复制次数的参数值修改为 0 或者负值，网页中复制次数会变为 3.

## 3.6 样例: HBase Aggregate

HBase Aggregate sample 通过创建一个新的表格并使用 coprocessor 对其进行统计分析，比如求最小最大值，计算行数，求和，求平均值，检验 HBase 的基本功能。你可以选择在 HBase shell 或者使用样例代码中提供的接口来检验某一统计功能。

你可以在随本文档一起提供的 examples.tar.gz 中找到此该样例的可运行代码。

### 3.6.1 样例代码

**关键类**

| 类名 | 描述 |
|---|---|
| com.intel.hbase.test.aggregate.AggregateOperationRunner | aggregate 方法的运行程序 |
| com.intel.hbase.test.aggregate.AggregateTest | 检测 aggregate 方法使用的时间 |
| com.intel.hbase.test.createTable.TableBuilder | 在 HDFS 中创建新的表格 |

**关键方法**

| 方法名称 | 描述 |
|---|---|
| com.intel.hbase.test.aggregate.AggregateOperationRunner:: getColumnInterpreter() | 转化列中的数据类型 |
| com.intel.hbase.test.aggregate.AggregateOperationRunner::run() | 提供统计功能，比如求最大最小值，求和等 |

**源代码**

下面的代码在 *com.intel.hbase.test.aggregate.AggregateOperationRunner* 类中.

```
private ColumnInterpreter<Long, Long> getColumnInterpreter() {
    String interpreter = props.getProperty(COLUMN_INTERPRETER_KEY);
    if (interpreter.equalsIgnoreCase("LongStr")) {
      return new LongStrColumnInterpreter();
```

```
    } else if (interpreter.equalsIgnoreCase("CompositeLongStr")) {
      String delim = props.getProperty(DELIM_KEY, ",");
      int index;
      index = Integer.parseInt(props.getProperty(INDEX_KEY, "0"));
      return new CompositeLongStrColumnInterpreter(delim, index);
    } else if (interpreter.equalsIgnoreCase("Long")) {
      return new LongColumnInterpreter();
    } else
      return null;
  }

public void run() {
    props = new Properties();
    try {
      InputStream in = new BufferedInputStream(new FileInputStream(
          configFileName));//get the configure input
      props.load(in);
    } catch (Exception e) {
      e.printStackTrace();
      System.out.println("File " + configFileName + " is NOT valid. Please have
a check first.");
      return;
    }

    String address = props.getProperty(HBASE_ADDRESS_KEY);//get the parameters
    byte[] table = Bytes.toBytes(props.getProperty(TABLE_KEY));
    String action = props.getProperty(ACTION_KEY);
    String timeoutValue = props.getProperty(RPC_TIMEOUT, "120000");

    Configuration conf = HBaseConfiguration.create();
    try {
      ZKUtil.applyClusterKeyToConf(conf, address);
      long timeout = Long.parseLong(timeoutValue);
      conf.setLong("hbase.rpc.timeout", timeout);
      //create the aggragation client
      AggregationClient aClient = new AggregationClient(conf);

      Scan scan = new Scan();
      if (!configScan(scan))
        return;

      final ColumnInterpreter<Long, Long> columnInterpreter =
                                                    getColumnInterpreter();
      if (columnInterpreter == null) {
```

```
      System.out.println("The value of 'interpreter' set in configuration file
      "+ configFileName + " seems wrong." + "Please have a check .");

      return;
      }

      try {//call the specified aggregate method

        if (action.equalsIgnoreCase("rowcount")) {
          Long rowCount = aClient.rowCount(table, columnInterpreter, scan);
          System.out.println("The result of the rowCount is " + rowCount);
        } else if (action.equalsIgnoreCase("max")) {
          Long max = aClient.max(table, columnInterpreter, scan);
          System.out.println("The result of the max is " + max);
        } else if (action.equalsIgnoreCase("min")) {
          Long min = aClient.min(table, columnInterpreter, scan);
          System.out.println("The result of the min is " + min);
        } else if (action.equalsIgnoreCase("sum")) {
          Long sum = aClient.sum(table, columnInterpreter, scan);
          System.out.println("The result of the sum is " + sum);
        } else if (action.equalsIgnoreCase("std")) {
          Double std = aClient.std(table, columnInterpreter, scan);
          System.out.println("The result of the std is " + std);
        } else if (action.equalsIgnoreCase("median")) {
          Long median = aClient.median(table, columnInterpreter, scan);
          System.out.println("The result of the median is " + median);
                          } else if (action.equalsIgnoreCase("avg")) {
                                Double avg = aClient.avg(table,
                                            columnInterpreter, scan);
                                System.out.println("The result of the avg is
                                                  " + avg);
        } else {
        System.out.println("The action '" + action + "' set in configuration file
                        "+ configFileName + " doesn't exist.");
          return;

        }
      } catch (Throwable e) {
        e.printStackTrace();
      }
    } catch (NumberFormatException e) {
      System.out.println("Invaild argument!");
    } catch (IOException e1) {
      e1.printStackTrace();
```

```
    }
}
```

### 3.6.2 使用

你可以在新建的表格上测试统计功能。无论是通过 HBase shell 还是样例代码中的接口。

#### 运行样例

1) 将包含样例代码的文件导入你已经装好英特尔® Hadoop 发行版 (IDH)的集群中。

2) 将在目录*/etc/hbase/conf/*下的文件 *hbase-site.xml* 拷贝到目录 *conf/*下。

3) 请确保样例代码的 lib 目录下的所有 jar 包与集群 jar 包保持一致，如果已经运行过本样例则可跳过这个步骤。

   检查*/root/usr/lib/*目录下的 jar 包，比如 hbase_VERSION.jar 和 zookeeper_VERSION.jar，若是它们的版本号和样例程序中的不同，你需要将不同的 jar 包拷贝到样例代码的 lib 目录下并且更改 build.xml 文件中的 jar 包名称。

4) 运行指令:

   ```
   ant BuildTable
   ```

   指令完成后，你可以进入 HBase shell 中检查新建的表格 *Test_Aggregate.*

   *你可以修改 BuildTable 目标的参数值来改变表格的大小，默认值为:*

   *7 false  7 表示插入个数数据条目 10^7*

5) 你可以

   - 输入指令: *hbase shell* ，然后运行统计功能能 (min, max, rowcount, sum, avg), 例如:

     ```
     aggregate 'max','Test_Aggregate','family:longStr2'
     ```

     ```
     hbase(main):009:0> aggregate 'max','Test_Aggregate','family:longStr2'
     The result of max for table Test_Aggregate is 9999
     0 row(s) in 0.1160 seconds
     ```

   修改文件 conf.co file 在目录*/usr/lib/hbase/examples/hbase/src/conf.co* 下来修改参数值，比如:

   //集群的 hbase 地址

   ```
   hbase_address=server-137.novalocal,server-138.novalocal
   ,server-139.novaloc:2181:/hbase
   ```

   //表格名称

   ```
   table=Test_Coprocessor
   ```

   //调用的统计功能( min, max, rowcount, sum, avg )

```
        action=max
    //表格列的数据类型
        columns=family:compositeLongStr
    //翻译类型包括 LongStr, CompositeLongStr, Long
        interpreter=CompositeLongStr
        delim=,
        index=1
        hbase.rpc.timeout=30000
```

然后，运行指令:

```
    ant Aggregate
```

```
build:

Aggregate:
    [java] /root/cj/hbaseTest/examples/hbase/.
    [java] The result of the max is 9999
    [java] ++ Time cost for Aggregation [config -> src/conf.co]: [552 ms] -->
0(h):0(m):0(s).552(ms) ++

BUILD SUCCESSFUL
```

注意：你需要根据你的集群中 ZooKeeper 的主机名来修改 hbase_address 参数

## 3.7 样例: HBase Parallel Scanning

HBase Replication sample 检测 HBase 上运行指令(rowcount, min, max, etc.)的有效性和可靠性。你可以在此例中比较 normal scanning 和 parallel scanning 的运行速度。

你可以在随本文档一起提供的 examples.tar.gz 中找到此该样例的可运行代码。

### 3.7.1 样例代码

**关键类**

| 类名 | 描述 |
| --- | --- |
| com.intel.hbase.test.aggregate.AggregateOperationRunner | aggregate 方法的运行程序 |

| | |
|---|---|
| com.intel.hbase.test.aggregate.AggregateTest | 检测 aggregate 方法使用的时间 |
| com.intel.hbase.test.createtable.TableBuilder | 在 HDFS 中创建新的表格 |
| com.intel.hbase.test.parallelscanner | 并行的启动扫描器 |
| com.intel.hbase.test.util.TimeCounter | 计算消耗的时间 |
| com.intel.hbase.test.util.TimeUtil | 时间工具的接口 |

### 关键方法

| 方法名称 | 描述 |
|---|---|
| com.intel.hbase.test.aggregate.AggregateOperationRunner::configScan() | 得到新的表格描述符 |
| com.intel.hbase.test.aggregate.AggregateOperationRunner:: getColumnInterpreter() | 通过数据类型分析列中数据 |
| com.intel.hbase.test.aggregate.AggregateTest::parseArgs() | 读取 conf.co 文件中的参数值 |
| com.intel.hbase.test.parallelscannel::main() | 并行扫描的主要方法 |

### 源代码

下面的代码在 com.intel.hbase.test.parallelscanner 类中.
其他关键方法代码可在上一样例中找到源代码。

```
package com.intel.hbase.test.parallelscanner;

import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.client.Result;
import org.apache.hadoop.hbase.client.ResultScanner;
import org.apache.hadoop.hbase.client.Scan;
import org.apache.hadoop.hbase.util.Bytes;
import com.intel.hbase.test.util.TimeCounter;

public class ParallelScannerTest {

    public static void main(String[] args) throws Exception {
        if (args.length < 4) {
          throw new Exception("Table name not specified.");
        }
        HTable table = new HTable(args[0]);//create a new table
        String startKey = args[1];//set the start key
        String stopKey = args[2]; //set the end key
        boolean isParallel = Boolean.parseBoolean(args[3]);
                  System.out.println("++ Parallel Scanning : " + isParallel + "
                                     ++");
```

```
        TimeCounter executeTimer = new TimeCounter();
        executeTimer.begin();//start counting the time elapsed
        executeTimer.enter();

        Scan scan = new Scan(Bytes.toBytes(startKey),
                        Bytes.toBytes(stopKey));
        int count = 0;
        ResultScanner scanner = isParallel ? table.getParallelScanner(scan) :
                                table.getScanner(scan);
        //get the normal or parallel scanner based on the args
        Result r = scanner.next();
        while (r != null) {//traverse the table
          count++;
          r = scanner.next();
        }
        System.out.println("++ Scanning finished with count : " + count + " ++");
        scanner.close();

        executeTimer.leave();//close the timer
        executeTimer.end();
        System.out.println("++ Time cost for scanning: " +
                           executeTimer.getTimeString() + " ++");
    }
}
```

### 3.7.2 使用

通过修改 conf.co 和 conf2.co 文件，你可以修改 aggregate 方法(例如 rowcounter,min,max 等)和集群中的其他参数来检测 HBase 的 parallel scanning 功能。参数的定义和上一个样例类似。

#### 运行样例

1) 将包含样例代码的文件导入你已经装好英特尔® Hadoop 发行版 (IDH)的集群中。
2) 将在目录*/etc/hbase/conf/*下的文件 *hbase-site.xml* 拷贝到目录 *conf/*下。
3) 请确保样例代码的 lib 目录下的所有 jar 包与集群 jar 包保持一致，如果已经运行过本样例则可跳过这个步骤。

   检查*/root/usr/lib/*目录下的 jar 包，比如 hbase_VERSION.jar 和 zookeeper_VERSION.jar，若是它们的版本号和样例程序中的不同，你需要将不同的 jar 包拷贝到样例代码的 lib 目录下并且更改 build.xml 文件中的 jar 包名称。
4) 首先你需要通过以下指令创建一个新的表格:

   ```
   ant BuildTable
   ```
5) 如果你想运行 normal scanning，只需键入如下指令:

```
ant NormalScan
```

6) 如果你想运行 parallel scanning，你可以修改 *build.xml* 中的参数，例如:

```
<target depends="build" name="ParallelScan" >

<java
classname="com.intel.hbase.test.parallelscanner.ParallelScannerTest"
failonerror="true" fork="yes">

      <classpath refid="test.classpath"/>

        <arg line="Test_Aggregate 0000,c01 9999,c01 true"/>

    </java>

</target>
```

Test_Aggregate: 表格名称

0000,c01: 开始键

9999,c01: 结束键

true: 开启 parallel scanning 功能

然后，运行指令:

```
ant ParallelScan
```

## 分析结果

执行完 normal scanning 后，你会得到如下结果:

```
build:

NormalScan:
    [java] ++ Parallel Scanning : false ++
    [java] ++ Scanning finished with count : 9999 ++
    [java] ++ Time cost for scanning: [775 ms] --> 0(h):0(m):0(s).299(ms) ++

BUILD SUCCESSFUL
Total time: 12 seconds
```

执行完 parallel scanning 后，你会得到如下结果:

```
build:

ParallelScan:
    [java] ++ Parallel Scanning : true ++
    [java] ++ Scanning finished with count : 9999 ++
    [java] ++ Time cost for scanning: [525 ms] --> 0(h):0(m):0(s).232(ms) ++
```

```
BUILD SUCCESSFUL
Total time: 12 seconds
```

可见 parallel scanning 比 normal scanning 速度更快.

## 3.8 样例: HBase Group-by

HBase Group-by sample 可以将表格中数据按照指定表达式要求分组，如果数据被分裂成几个小部分，该指令可以运行的更迅速。

你可以在随本文档一起提供的 examples.tar.gz 中找到此该样例的可运行代码。

### 3.8.1 样例代码

**关键类**

| 类名 | 描述 |
|---|---|
| com.intel.hbase.table.test.groupby.GroupByOperationRunner | Group-by 方法的运行启动函数 |
| com.intel.hbase.table.test.groupby.GroupByTest | 测试 group-by 消耗的时间 |

**关键方法**

| 方法名称 | 描述 |
|---|---|
| com.intel.hbase.table.test.groupby. GroupByOperationRunner:: configGroupByKey () | 按键值将列值分组 |
| com.intel.hbase.table.test.groupby. GroupByOperationRunner::configSelectExpressions() | 运行统计指令 |
| com.intel.hbase.table.test.groupby. GroupByTest::parseArgs() | 读取 conf.gb 中的参数值 |

**源代码**

下面的代码在 *com.intel.hbase.table.create.GroupByOperationRunner* 类中.

```
  private boolean configGroupByKey(List<Expression> groupByExpresstions,
                                   List<Expression> selectExpresstions) {
    String columns = props.getProperty(GROUGBY_KEY_KEY);
    StringTokenizer st = new StringTokenizer(columns, ", ");
    while (st.hasMoreTokens()) {//traverse all the columns
      //get the columns by using the split token ":"
      String column[] = st.nextToken().split(":");
      switch (column.length) {
      case 2:
//store the value of the columns in the array, groupByExpressions
      groupByExpresstions.add(ExpressionFactory.columnValue(column[0],column
[1]));
//add the columns grouped by key
```

```
      selectExpresstions.add(ExpressionFactory.groupByKey(ExpressionFactory.c
      olumnValue(column[0], column[1])));
        break;
      default:
        System.out.println("The value of 'KEY' set in configuration file "
            + configFileName + " seems incorrect. Please have a check first.");
        return false;
      }
    }
    return true;
  }


 private boolean configSelectExpresstions(List<Expression> selectExpresstions)
{
    String aggregations = props.getProperty(GROUGBY_SELECT_KEY);
    StringTokenizer st = new StringTokenizer(aggregations, ", ");
    while (st.hasMoreTokens()) {
      String aggregation[] = st.nextToken().split("#");
      String action = aggregation[0];
      String columns = aggregation[1];
      String column[] = columns.split(":");
      if(2 != column.length){
        System.out.println("The value of 'SELECT' set in configuration file "
            + configFileName + " seems incorrect. Please have a check first.");
        return false;
      }else{
        Expression columnValueExp = ExpressionFactory.columnValue(column[0],
                                                                  column[1]);
        //select the column value according to the action
        if (action.equalsIgnoreCase("rowcount")) {
          selectExpresstions.add(ExpressionFactory.count(columnValueExp));
        } else if (action.equalsIgnoreCase("max")) {
          selectExpresstions.add(ExpressionFactory.max(columnValueExp));
        } else if (action.equalsIgnoreCase("min")) {
          selectExpresstions.add(ExpressionFactory.min(columnValueExp));
        } else if (action.equalsIgnoreCase("sum")) {
          selectExpresstions.add(ExpressionFactory.sum(columnValueExp));
        } else if (action.equalsIgnoreCase("std")) {
          selectExpresstions.add(ExpressionFactory.stdDev(columnValueExp));
        } else if (action.equalsIgnoreCase("avg")) {
          selectExpresstions.add(ExpressionFactory.avg(columnValueExp));
        } else{
        }
```

```
        }
    }
    return true;
}
```

### 3.8.2 使用

你可以在此样例中检测分组功能的有效性。

#### 运行样例

1) 将包含样例代码的文件导入你已经装好英特尔® Hadoop 发行版 (IDH)的集群中。

2) 将在目录*/etc/hbase/conf/*下的文件 *hbase-site.xml* 拷贝到目录 *conf/*下。

3) 请确保样例代码的 lib 目录下的所有 jar 包与集群 jar 包保持一致，如果已经运行过本样例则可跳过这个步骤。

   检查*/root/usr/lib/*目录下的 jar 包，比如 hbase_VERSION.jar 和 zookeeper_VERSION.jar，若是它们的版本号和样例程序中的不同，你需要将不同的 jar 包拷贝到样例代码的 lib 目录下并且更改 build.xml 文件中的 jar 包名称。

4) 检查集群中是否已存在 Test_Aggregate 表格，避免已有数据被删除。

5) 执行指令:

   ```
   ant BuildTable
   ```

   指令完成后，你可以进入 HBase shell 中检查新建的表格 *Test_Aggregate.*

   *你可以修改 BuildTable 目标的参数值来改变表格的大小，默认值为:*

   *7 false  7 表示插入个数数据条目 10^7*

6) 现在你可以检测 HBase 的分组功能:

   ```
   ant GroupBy
   ```

   注意：你可以修改 conf.gb 文件中的参数值，例如：

   ```
   //集群中 hbase 的地址
   hbase_address=bdqac2-node2,bdqac2-node3,bdqac2-node4:2181:/hbase
   //表格名称
   table=Test_Aggregate
   hbase.rpc.timeout=120000
   //关键值的类型
   ```

```
KEY=family:mod10Str

  // 调用的统计方法( min, max, rowcount, sum, avg ) 作用于列簇上

SELECT=rowcount#family:longStr2
```

### 分析结果

如下为分组功能运行消耗时间:

```
build:

GroupBy:
    [java] /root/mtest/HBaseSampleCaseTest/.
    [java] ++ Time cost for GroupBy [config -> src/conf.gb]: [795 ms] -->
0(h):0(m):0(s).795(ms) ++
```

## 3.9 样例: HBase Expressionfilter

HBase Expressionfilter sample 中定义了表达式，根据此表达式过滤数据得到结果集。

你可以在随本文档一起提供的 examples.tar.gz 中找到此该样例的可运行代码。

### 3.9.1 样例代码

**关键类**

| 类名 | 描述 |
|---|---|
| com.intel.hbase.table.test.expressionfilter.ExpressionFilterTest | 测试表达式过滤方法的有效性 |

**关键方法**

| 方法名称 | 描述 |
|---|---|
| com.intel.hbase.table.test.expressionfilter.ExpressionFilterTest::main() | 测试中的主要方法 |

### 源代码

如下代码在 *com.intel.hbase.table.test.expressionfilter.ExpressionFilterTest* 类中.

```
public class ExpressionFilterTest {

  public static void main(String[] args) throws Exception {
      if (args.length < 2) {
        throw new Exception("Table name not specified.");
      }
```

```
        Configuration conf = HBaseConfiguration.create();
        HTable table = new HTable(conf,args[0]);
        String startKey = args[1];

        //set the time counter
        TimeCounter executeTimer = new TimeCounter();
        executeTimer.begin();
        executeTimer.enter();

//define the expression used to filter the data in the table. The expression
here is to filter out the data that equals to 99.
         Expression exp = ExpressionFactory.eq(ExpressionFactory.toLong(
         ExpressionFactory.toString(ExpressionFactory.columnValue("family",
         "longStr2"))), ExpressionFactory.constant(Long.parseLong("99")));
         ExpressionFilter expressionFilter = new ExpressionFilter(exp);
         Scan scan = new Scan(Bytes.toBytes(startKey), expressionFilter);
         int count = 0;
         ResultScanner scanner = table.getScanner(scan);
         Result r = scanner.next();
       //scan the table
         while (r != null) {
           count++;
           r = scanner.next();
         }
         System.out.println("++ Scanning finished with count : " + count + " ++");
         scanner.close();

         executeTimer.leave();
         executeTimer.end();
         System.out.println("++ Time cost for scanning: " +
executeTimer.getTimeString() + " ++");
    }
}
```

### 3.9.2 使用

你可以在代码中定义自己的表达式来检测 HBase 的 expressionfilter 功能点。

#### 运行样例

1) 将包含样例代码的文件导入你已经装好英特尔® Hadoop 发行版 (IDH)的集群中。

2) 将在目录*/etc/hbase/conf/*下的文件 *hbase-site.xml* 拷贝到目录 *conf/*下。

3) 请确保样例代码的 lib 目录下的所有 jar 包与集群 jar 包保持一致，如果已经运行过本样例则可跳过这个步骤。

检查*/root/usr/lib/*目录下的 jar 包，比如 hbase_VERSION.jar 和 zookeeper_VERSION.jar，若是它们的版本号和样例程序中的不同，你需要将不同的 jar 包拷贝到样例代码的 lib 目录下并且更改 build.xml 文件中的 jar 包名称。

4) 执行指令:

```
ant BuildTable
```

指令完成后，你可以进入 HBase shell 中检查新建的表格 *Test_Aggregate.*

*你可以修改 BuildTable 目标的参数值来改变表格的大小，默认值为:*

*7 false 7 表示插入个数数据条目 10^7*

注意：你也可以通过修改 build.xml 中的表格名来使用自己的表格

5) 现在你可以执行指令:

```
ant ExpressionFilter
```

### 分析结果

```
build:

ExpressionFilter:
    [java] ++ Scanning finished with count : 1 ++
    [java] ++ Time cost for scanning: [329 ms] --> 0(h):0(m):0(s).211(ms) ++

BUILD SUCCESSFUL
Total time: 12 seconds
```

## 3.10 样例: HBase MultiRowRangeFilter

HBase MultiRowKeyRangeFilter sample 中定义了多个需要扫描的 rowkey 区段，其效果类似于 Scan 的 startRow 以及 stopRow，但能同时设置多组。

你可以在如下目录中找到该样例的代码：
/usr/lib/hbase/examples/

### 3.10.1 样例代码

#### 关键类

***HBase***

| 类名 | 描述 |
| --- | --- |
| com.intel.hbase.test.multirowrangefilter.<br>MultiRowRangeFilterTest | 测试 MultiRowRangeFilter 的有效性 |

### 关键方法

| 方法名称 | 描述 |
| --- | --- |
| com.intel.hbase.table.test.multirowrangefilter.MultiRowRangeFilterTest::main() | 测试中的主要方法 |

### 源代码

如下代码在 *com.intel.hbase.test.multirowrangefilter.MultiRowRangeFilterTest* 类中。

```
public static void main(String[] args) throws Exception {
    if (args.length < 1) {
      throw new Exception("Table name not specified.");
    }
    Configuration conf = HBaseConfiguration.create();
    HTable table = new HTable(conf, args[0]);

    TimeCounter executeTimer = new TimeCounter();
    executeTimer.begin();
    executeTimer.enter();
    Scan scan = new Scan();
    List<RowKeyRange> ranges = new ArrayList<RowKeyRange>();
    ranges.add(new RowKeyRange(Bytes.toBytes("001"),
Bytes.toBytes("002")));
    ranges.add(new RowKeyRange(Bytes.toBytes("003"),
Bytes.toBytes("004")));
    ranges.add(new RowKeyRange(Bytes.toBytes("005"),
Bytes.toBytes("006")));
    Filter filter = new MultiRowRangeFilter(ranges);
    scan.setFilter(filter);
    int count = 0;
    ResultScanner scanner = table.getScanner(scan);
    Result r = scanner.next();
    while (r != null) {
      count++;
      r = scanner.next();
    }
    System.out.println("++ Scanning finished with count : " + count + "
++");
    scanner.close();

    executeTimer.leave();
    executeTimer.end();
```

```
    System.out.println("++ Time cost for scanning: "
        + executeTimer.getTimeString() + " ++");
  }
```

### 3.10.2 使用

你可以在代码中定义自己的区段来检测 HBase MultiRowRangeFilter 的功能点。

#### 运行样例

1) 将包含样例代码的文件导入你已经装好英特尔® Hadoop 发行版 (IDH)的集群中。

2) 将在目录*/etc/hbase/conf/*下的文件 *hbase-site.xml* 拷贝到目录 *conf/*下。

3) 请确保样例代码的 lib 目录下的所有 jar 包与集群 jar 包保持一致，如果已经运行过本样例则可跳过这个步骤。

   检查*/root/usr/lib/*目录下的 jar 包，比如 hbase_VERSION.jar 和 zookeeper_VERSION.jar，若是它们的版本号和样例程序中的不同，你需要将不同的 jar 包拷贝到样例代码的 lib 目录下并且更改 build.xml 文件中的 jar 包名称。

4) 执行指令:

   ```
   ant BuildTable
   ```

   指令完成后，你可以进入 HBase shell 中检查新建的表格 *Test_Aggregate*.

   你可以修改 *BuildTable* 目标的参数值来改变表格的大小，默认值为：

   7 false  7 表示插入个数数据条目 10^7

   注意：你也可以通过修改 build.xml 中的表格名来使用自己的表格

5) 现在你可以执行指令:

   ```
   ant MultiRowRangeFilter
   ```

#### 分析结果

```
build:

MultiRowRangeFilter:
     [java] ++ Scanning finished with count : 30 ++
     [java] ++ Time cost for scanning: [33 ms] --> 0(h):0(m):0(s).33(ms)
++

BUILD SUCCESSFUL
Total time: 11 seconds
```

## 3.11 样例: 全文索引

### 3.11.1 目的

为 HBase 建立全文索引。

### 3.11.2 术语解释

Lucene: Apache 项目，开源的全文检索工具包。

Document: Lucene 索引中的一条记录。

Field: 一个 Document 可以拥有多个 fields。每个 field 拥有各自的名称和值，名称唯一标识这个 field，值则用来建立索引。

### 3.11.3 实现方法

索引的建立和搜索都基于 Lucene

索引储存于 hdfs

### 3.11.4 入门

全文索引是在何时和如何建立的？我们监听每个 region 上的数据更新(add/delete/WALRestore/split)，同时同步的建立索引(add/delete/WALRestore/split)。

这些索引被存储在不同的分片(shard)上，每一个索引分片与 region 是一一对应的。

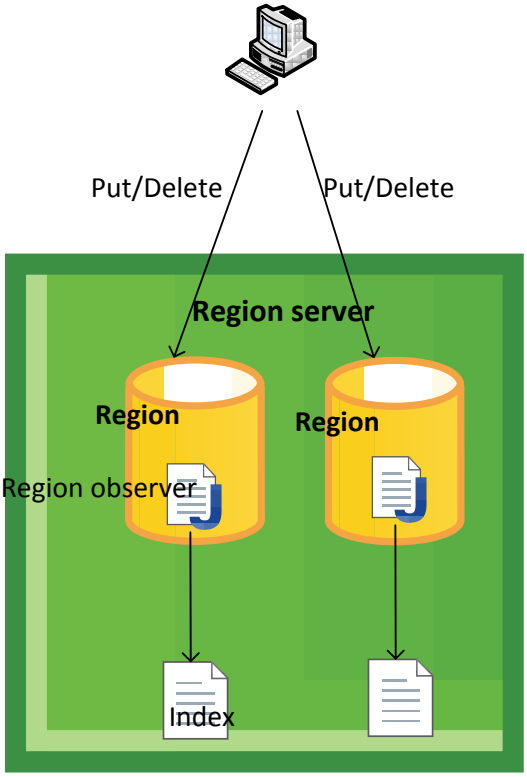
**请注意：**

HBase 的 add/delete/WALRestore/split 操作会被监听并更新索引。

若使用 HBase 的 put 对 HBase 中的同一条数据进行更新，已有的索引记录并不会被更新，只是新增一条关于该 HBase 数据的索引记录。如果有这种对 HBase 中同一条数据进行更新的操作需求，请使用 HBase 的 delete 和 put 操作。

如果索引的创建已经被启用，HTable 的 checkAndPut, checkAndDelete, increment, incrementColumnValue，和 append 方法将不被支持。

下面的图片显示的索引是如何工作的。




## 3.11.5 类、属性介绍

### 3.11.5.1 IndexMetadata

我们在 HTableDescriptor 上定义一个名为 IndexMetadata 的属性

- 如果这个元数据被赋值，当记录被 put/deleted/WALReloaded/Split 时，索引将作出相应的操作。
- 如果这个元数据没有被赋值，那么索引将不会被建立。

IndexMetadata 描述了 Lucene 上的 fields 与 HBase 上的列之间的映射

- Update mode：是否采用更新模式，目前只有非更新模式被支持。**该值不需要被额外设置**。
- Region index policy：这个属性定义了索引是如何被建立的，现在仅有 synchronous index policy (SynchronousRegionIndexPolicy)（同步索引）被实现。**该值不需要进行额外设置**。
- 一个或多个 IndexFieldMetadata：IndexFieldMetadata 是 Lucene Field 和 HBase 列之间映射被实际定义的地方。
  - Name：这是索引 field 的名字
  - 一个或多个 IndexedHBaseColumn：一个 Lucene field 可能包含一个或多个 HBase 的列，不同的列的值被一个 char 类型的值所分割，例如：**“field value = column value + separator + column value + separator + column value …”**

    （1） Family name： Hbase 的 family

    （2） Qualifier names：Hbase 的 qualifier

- Separator:在 field 值里，用于分隔来自于不同 HBase 列的值，默认是逗号。
- Type:这个 field 的值类型
- Stored：表明这个 field 的值是否被保存
- Indexed:表明这个 field 是否需要建立索引
- Tokenized：表明这个 field 是否需要进行分词
- Analyzer name:目前已有七个 analyzer 被实现。每个 analyzer 包含了对于当前 Lucene field 进行分词和检索的方式，信息。
- The arguments of the analyzer:一个 analyzer 可能有参数，如果有，直接对该属性赋值。后面的 sample 中会介绍如何使用。

- Search mode:目前，我们提供三种搜索的 mode，分别是 NearRealTime, LastBufferFlush and LastCommit。我们将在搜索章节讨论它们

### 3.11.5.2 Analyzers

目前，我们对 Analyzer 有七种实现。一旦 analyzer 被确定，那么查询的方式也会随之确定。

- *WHITE_SPACE_ANALYZER*
- *STOP_ANALYZER*
- *SEPARATOR_ANALYZER*
- *COMMA_SEPARATOR_ANALYZER*
- *MMSEG_MAX_WORD_ANALYZER*
- *MMSEG_COMPLEX_ANALYZER*
- *SIZABLE_NGRAM_ANLAYZER*

#### 1）MMSEG_MAX_WORD_ANALYZER

该 Anlayzer 使用字典对中文进行分词，对于英文将直接使用 Lucene 的 StopAnalyzer 进行分词。字典是可以可配置的。

代码如下，你可以在 IndexFieldMetadata 中使用这个 analyzer

```
String dicPath = "";
fieldMetadata.setAnalyzerMetadataName(IndexAnalyzerNames.MMSEG_MAX_WO
RD_ANALYZER);
AnalyzerMetadataArg[] args = new AnalyzerMetadataArg[]{new
AnalyzerMetadataArg(dicPath, String.class)};
fieldMetadata.setAnalyzerMetadataArgs(args);
```

如果 AnalyzerMetadataArg 没有被设置，那么将使用默认字典，代码如下。

```
fieldMetadata.setAnalyzerMetadataName(IndexAnalyzerNames.MMSEG_MAX_WO
RD_ANALYZER);
```

#### 2）MMSEG_COMPLEX_ANALYZER

这个 Analyzer 与 *MMSEG_MAX_WORD_ANALYZER* 是几乎相同的，其中 *MMSEG_MAX_WORD_ANALYZER* 会把中文分词分割到更小的粒度。

#### 3）SEPARATOR_ANALYZER

这 analyzer 直接用字符去分割文本进行分词。默认的分隔符是逗号。

```
fieldMetadata.setAnalyzerMetadataName(IndexAnalyzerNames.SEPARATOR_AN
ALYZER);
AnalyzerMetadataArg[]args = new AnalyzerMetadataArg[]    {      new
AnalyzerMetadataArg(Character.valueOf('-'), Character.class)};
fieldMetadata.setAnalyzerMetadataArgs(args);
```

### 4）COMMA_SEPARATOR_ANALYZER

这个是 SEPARATOR_ANALYZER 的子类，在这个 analyzer 里，将使用逗号作为分隔符。

```
fieldMetadata.setAnalyzerMetadataName(IndexAnalyzerNames.COMMA_SEPARA
TOR_ANALYZER);
```

### 5）WHITE_SPACE_ANALYZER

这个 analyzer 使用 Lucene 的 WhitespaceAnalyzer 进行分词，使用空格来分割文本。

```
fieldMetadata.setAnalyzerMetadataName(IndexAnalyzerNames.WHITE_SPACE_
ANALYZER);
```

### 6）STOP_ANALYZER

这个 analyzer 使用 Lucene 的 StopAnalyzer 进行分词。

```
fieldMetadata.setAnalyzerMetadataName(IndexAnalyzerNames.STOP_ANALYZE
R);
```

### 7）SIZABLE_NGRAM_ANALYZER

这个 analyzer 将文本生成若干 token，每个 token 都拥有固定的长度。这样不仅节省了磁盘空间，同时加快了搜索的性能。

下面的代码显示了如何使用这个 analyzer。

如下，使用默认的 *SIZABLE_NGRAM_ANLAYZER*，逗号将被作为分隔符，token 长度被设置为{1,3,7}，这意味着经过分词过后的 token 的长度将是 1,3 或者 7。

```
fieldMetadata.setAnalyzerMetadataName(IndexAnalyzerNames.SIZABLE_NGRA
M_ANLAYZER);
```

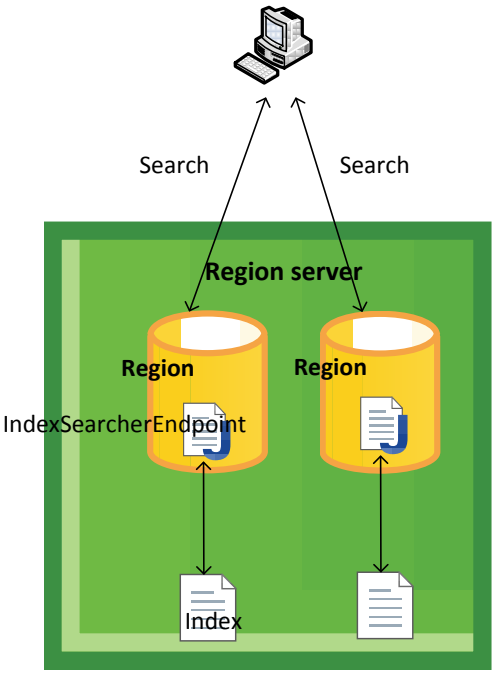
如下，使用可配置的 analzyer，在这个例子中，“-”被用来作为分隔符，int 类型数组{1,5,11}被用来指明分词过后的 token 的长度。

```
int[] tokenLength = new int[]{1,5,11};

fieldMetadata.setAnalyzerMetadataName(IndexAnalyzerNames.SIZABLE_NGRA
M_ANLAYZER);

AnalyzerMetadataArg[]   args   =   new   AnalyzerMetadataArg[]   {   new
AnalyzerMetadataArg(Character.valueOf('-'),   Character.class),   new
AnalyzerMetadataArg(tokenLength, tokenLength.getClass()) };

fieldMetadata.setAnalyzerMetadataArgs(args);
```

### 3.11.6 搜索

如何搜索这些分布式的索引？我们使用 HBase 的 endpoint 来将这些搜索请求分发给各个 regions。每一个 region 负责处理自己的搜索请求，并将这些结果返回给客户端。客户端会对这些结果做必要的合并，排序和分页工作。

如下的图片显示了搜索功能是如何工作的




现在我们提供三种搜索模式，*NearRealTime*, *LastBufferFlush* and *LastCommit*。这个方式在 IndexMetadata 里被设置。

- *NearRealTime*: 实时搜索模式，你可以在索引中进行 near real time 搜索。但是在大数据量插入的情况下，这种搜索的性能将受到影响，同时也会生成很多琐碎的索引段，从而对建立索引产生影响。
- *LastBufferFlush:* 这个搜索模式不会像 *NearRealTime* 那样搜索实时数据，但是这种模式会比 *NearRealTime* 拥有更好的搜索性能同时对建立索引产生更小的影响。
- *LastCommit:* 这种搜索模式只能搜索到那些自上次 HBase flush 之前所产生的数据。实时性比 *LastBufferFlush* 更差。

### 3.11.7 搜索 API

搜索 API 定义如下。

第一个 API 将请求分配给所有运行的 region。

第二个 API 只将搜索请求分配给那些 start key 在参数 startKey 和 endKey 之间的 region。

请注意：第一个构造方法省去了通过 HTable 读取 IndexMetadata 的过程，所以拥有更好的性能。

```
public class IndexSearcherClient {
 public IndexSearcherClient(HTable table, IndexMetadata indexMetadata){
  }
  public IndexSearcherClient(HTable table) throws IOException {
  }
  Public IndexSearchResponse search(QueryExpression queryExpression,
int start, int count, IndexSort sort)
  Public IndexSearchResponse search(QueryExpression queryExpression,
int start, int count, IndexSort sort, IndexScan scan)
 Public IndexSearchResponse search(QueryExpression
queryExpression,byte[] startKey, byte[] endKey, int start, int count,
IndexSort sort, IndexScan scan)
}
```

### 3.11.8 搜索 API 的参数列表

1） **QueryExpression**

QueryExpression，这是一个只包含名字和值的对象，名字是 Lucene field 的名字，值是搜索关键词。

现在我们有几种不同的 QueryExpression：

MatchQueryExpression:查询该 field 上所有精确匹配搜索关键词的结果。

StartWithQueryExpression:查询该 field 上所有以该关键词开始的结果。

EndWithQueryExpression: 查询该 field 上所有以该关键词结束的结果。

ContainQueryExpression: 查询该 field 上所有包含该关键词的结果。

RangeQueryExpression:查询该 field 上所有符合该范围的结果。这个查询有以下属性：

- Name: 索引 field 的名字
- Lower value: 该 range 的下边界，如果这个值是空，则指这个查询没有下边界
- includeLower: 这是一个布尔值，它表明这个 range 是否包含 Lower value(> or >= lower value)
- Upper value: 该 range 的上边界，如果这个值是空，则指这个查询没有上边界
- includeUpper: 这是一个布尔值，它表明这个 range 是否包含 Upper value (< or <= upper value).

AndQueryExpression: 这是一个布尔型的与查询，它可以包含所有 QueryExpression 类型的实例。

OrQueryExpression: 这是一个布尔型的或查询，它可以包含所有 QueryExpression 类型的实例。

2） **IndexSort**

这个参数是用来对结果进行排序的。如果这个值是空，则结果会被按照关联度排序

下面的代码是 IndexSort 的构造函数

```
public IndexSort()
public IndexSort(IndexSortField... sortFields)
```

它有一个空的构造函数和一个包含若干 IndexSortFields 的构造函数

- 如果使用空构造函数，那么结果会被按照关联度排序
- 如果多个 IndexSortField 被使用，那么结果会按照这些 IndexSortField 的顺序进行排序

IndexSortField：

在 IndexSortField 中有两个预定义的实例，`IndexSortField.FIELD_SCORE` 和 `IndexSortField.FIELD_DOC`，分表表示按照关联度和按照索引被建立的顺序进行排序。

```
public IndexSortField(String fieldName, boolean reverse,IndexSortPolicy
policy)
```

目前共有三种 IndexSortPolicy。

- *SCORE*: 按照关联度对结果排序，这是一个默认的排序策略
- *DOC*: 按照索引被建立的顺序对结果进行排序
- *FIELD*: 按照 field 的值对结果排序。如果你希望使用 field 来排序，请确保该 field 是被储存的。

#### 3） IndexScan

该参数指定了，有哪些 family/qualifier 需要从服务端被返回。如果该参数是 null，那么所有的 family/qualifier 都将被返回。

#### 4） 其他参数

`start` 和 `count` 这两个参数是用来分页的。表示希望取得第 `start` 到 `start+count` 之间的数据。

startKey 和 endKey，用来缩小搜索范围，如果您知道搜索结果的 row key 范围，那么这两个参数就会比较有用。

#### 5） 返回值

返回值是一个 `IndexSearchResponse` 的实例，它包含了以下属性

- `IndexableRecord` 的数组: `IndexableRecord` 与索引的 Document 有一一对应的关系。
  - Id: 这是 Lucene Document 的 id，Bytes.toBytes(id)便是 `HBase` 中的一条记录的 `row key`。
  - Result: 这是 HBase 的 Result 对象。是与该索引记录对应的 HBase 数据的查询结果。
  - A map of stored fields: 该 map 的键值是 String 类型的，它是 field 的名字，值是 Object 类型，它是 field 的值。该 map 包含了所有在 `IndexFieldMetadata` 中被标记为存储的 fields
- Total hits: 表示在索引中有多少条数据被该搜索命中。

### 3.11.9 样例

你可以在如下目录中找到该样例的代码：

/usr/lib/hbase/examples/

下面的代码片段显示了如何创造一个拥有 IndexMetadata.的 HTable

```
public static void main(String[] args) throws IOException {
```

```
  if (args.length <1){
    System.exit(1);
  }
  String tableName = args[0];
    Configuration conf = HBaseConfiguration.create();
    IndexMetadata indexMetadata = new IndexMetadata();
    indexMetadata.setSearchMode(SearchMode.LastCommit);
    IndexFieldMetadata valueMeta1 = new IndexFieldMetadata();
    valueMeta1.setName(Bytes.toBytes("field1"));
    valueMeta1
        .setAnalyzerMetadataName(IndexAnalyzerNames.COMMA_SEPARATOR_AN
ALYZER);
    valueMeta1.setIndexed(true);
    valueMeta1.setTokenized(true);
    valueMeta1.setStored(true);
    valueMeta1.setType(IndexFieldType.STRING);
    IndexedHBaseColumn column11 = new IndexedHBaseColumn(
        Bytes.toBytes("values"));
    column11.addQualifierName(Bytes.toBytes("value"));
    valueMeta1.addIndexedHBaseColumn(column11);
    indexMetadata.addFieldMetadata(valueMeta1);
    HTableDescriptor desc = new HTableDescriptor(tableName);
    desc.setIndexMetadata(indexMetadata);
    desc.addFamily(new HColumnDescriptor("values"));
    HBaseAdmin admin = null;
    try {
      admin = new HBaseAdmin(new Configuration(conf));
      admin.createTable(desc);
    } finally {
      admin.close();
    }
  }
```

下面的代码片段表明了如何使用这些 API 来搜索

```
public static void main(String[] args) throws Throwable{
    if (args.length <1){
        System.exit(1);
    }
    String tableName = args[0];
    Configuration conf = HBaseConfiguration.create();
    HTable table = new HTable(conf, Bytes.toBytes(tableName));
    QueryExpression q = new MatchQueryExpression("field1", "Im value1");
    IndexSearcherClient client = new IndexSearcherClient(table);
    IndexSearchResponse resp = client.search(q, 0, 50, null);
    IndexableRecord[]records = resp.getRecords();
```

```
    if(records!=null) {
    for(IndexableRecord record : records) {
      Result result = record.getResult();
      byte[] rowKey = result.getRow();
      String showRowKey = Bytes.toString(rowKey);
     byte[]    value    =    result.getValue(Bytes.toBytes("values"),
Bytes.toBytes("value"));
      StringBuilder sb = new StringBuilder();

  sb.append("rowkey :").append(showRowKey).append("value :").append(B
ytes.toString(value));
      System.out.println(sb.toString());
    }
    }
  }
```

## 运行样例

1) 将包含样例代码的文件导入你已经装好英特尔® Hadoop 发行版 (IDH)的集群中。

2) 将在目录*/etc/hbase/conf/*下的文件 *hbase-site.xml* 拷贝到目录 *conf/*下。

3) 请确保样例代码的 lib 目录下的所有 jar 包与集群 jar 包保持一致，如果已经运行过本样例则可跳过这个步骤。

   检查*/root/usr/lib/*目录下的 jar 包，比如 hbase_VERSION.jar 和 zookeeper_VERSION.jar，若是它们的版本号和样例程序中的不同，你需要将不同的 jar 包拷贝到样例代码的 lib 目录下并且更改 build.xml 文件中的 jar 包名称。

4) 执行指令:

   ```
   ant CreateTable
   ```

   指令完成后，你可以进入 HBase shell 中检查新建的表格

```
hbase(main):004:0> describe 'Test_fullTextIndex'
DESCRIPTION                                          ENABLED
 {NAME => 'Test_fullTextIndex', coprocessor$1 => '|org true
 .apache.hadoop.hbase.search.LuceneRegionCoprocessor|1
 073741823|', FAMILIES => [{NAME => 'values', DATA_BLO
 CK_ENCODING => 'NONE', BLOOMFILTER => 'NONE', REPLICA
 TION_SCOPE => '0', VERSIONS => '3', COMPRESSION => 'N
 ONE', MIN_VERSIONS => '0', TTL => '2147483647', KEEP_
 DELETED_CELLS => 'false', BLOCKSIZE => '65536', IN_ME
 MORY => 'false', ENCODE_ON_DISK => 'true', BLOCKCACHE
  => 'true'}], index metadata => org.apache.hadoop.hba
 se.search.IndexMetadata:{version:0, searchManner:Last
 Commit, update:false,fieldMetadata:[{field1:org.apach
 e.hadoop.hbase.search.IndexMetadata$IndexFieldMetadat
 a:{name:field1, type:STRING,analyzerMetadataName:org.
 apache.hadoop.hbase.search.IndexContext$CommaSeparato
 rAnalyzerMetadata[],indexed:true,stored:true,tokenize
 d:true,separator:,,columns:[{values:org.apache.hadoop
 .hbase.search.IndexMetadata$IndexedHBaseColumn:{famil
 yName:values, qualifierNames[value,]}},]}},]}}
1 row(s) in 0.0540 seconds
```

现在可以往表格中插入若干数据，以方便测试：

```
hbase(main):004:0> scan  'Test_fullTextIndex'
ROW                  COLUMN+CELL
 1001                column=values:value, timestamp=1353984097366, value=Im val
                     ue1
 1002                column=values:value, timestamp=1353984104133, value=Im val
                     ue1
 1003                column=values:value, timestamp=1353984112350, value=Im val
                     ue2
3 row(s) in 0.0320 seconds
```

一共有三行记录，其中，有两个位于索引(values:value)值都是”Im value1”。

由于我们采用的是非实时搜索模式，所以必须在 hbase shell 里 flush 该表：

```
hbase(main):001:0> flush 'Test_fullTextIndex'
0 row(s) in 0.8580 seconds
```

现在退出 hbase shell。可以执行指令：

```
ant Search
```

```
     [java] rowkey :1002value :Im value1
     [java] rowkey :1001value :Im value1

BUILD SUCCESSFUL
```

我们可以看到，值为”Im value1”的两行被搜索了出来。

## 3.11.10 如何更新索引结构

当一个表的索引结构需要调整的时候，比如说为表添加需要索引的列，该如何实现呢？
只需要简单的更新该 HTable 的 HTableDescriptor 中定义的 IndexMetadata。步骤如下：
Disable 该 HTable。
更新它的 HTableDescriptor 的 IndexMetadata。
Enable 该 HTable。

接下来该列上新插入的数据就能建立全文索引了。

代码如下所示。

```
Configuration conf = HBaseConfiguration.create();
HBaseAdmin admin = null;
try {
admin = new HBaseAdmin(new Configuration(conf));
  HTableDescriptor desc = admin.getTableDescriptor(tableName);
      IndexMetadata indexMetadata = desc.getIndexMetadata();
      // 你可以在此处修改 indexMetadata，添加你想增加的列，或者不再需要的列。
      // 此处为 indexMetadata 添加/删除列的代码被省略。
      // 然后将改变过后的 indexMetadata 重新赋值给 HTableDescriptor。
      desc.setIndexMetadata(indexMetadata);
      admin.disableTable(tableName);
      admin.modifyTable(tableName, desc);
      admin.enableTable(tableName);
} finally {
  if(admin != null){
admin.close();
}
}
```

# 4 ZooKeeper

## 4.1 前提

确定 Hadoop，ZooKeeper 已经被正确安装，配置，并且正在运行。

## 4.2 概要

作为一个支持分布式系统同步软件，ZooKeeper 可以维护系统配置，集群用户信息，命名以及其他相关信息。

ZooKeeper 将不同同步服务集成到一个简单易用的用户界面。它具有分布性和高可靠性。

ZooKeeper 中有两种运行模式：

- 单机模式：

  ZooKeeper 仅运行在一个服务器上。可用来测试相关环境配置。

- 复制模式：

ZooKeeper 运行在集群上，适合生产模式。这个计算机集群被称为一个“集合体”。服务器的数目应为单数。

ZooKeeper 为一树状文件系统，有 znode 组成。每个 znode 均与 ACL(Access Control List)相连。

ZooKeeper 中约有九种操作：

| 操作 | 描述 |
|---|---|
| Create | Create a znode (a parent node must be provided) |
| Delete | Delete a znode, which has no children nodes |
| Exists | Test if the znode exists |
| getACL,setACL | Get/set the ACL of a znode |
| getChildren | Get the list of children nodes of a znode |
| getData,setData | Get/set the stored data of a znode |
| Sync | Synchronize the view of znode client with ZooKeeper |

客户端可以在 znode 上设置 watch。如果 znode 发生改变，watch 会被一次触发，然后被清空。例如，某客户端调用 getData (“/znode1”, true)，使得/znode1 的数据被更改，客户端会得到对于/znode1 的一次 watch event。如果/znode1 再次改变，除非客户端又一次进行读取操作且设置一个新的 watch，否则不会产生 watch event。

ZooKeeper 中使用 ACL 操作来控制对于 znode 的读写路径。它没有 znode 的拥有者概念。特别的，它的操作不会付诸于孩子节点。例如，若/a 仅被 ip: 127.0.0.1 可读，且/a/b 具有普遍可读性，那么你就可以读取/a/b。ACL 操作非递归。

ZooKeeper 支持 5 种权限：

- CREATE: 创建一个子节点
- READ: 得到该节点数据并列出子节点
- WRITE: 设置节点数据
- DELETE: 删除一个子节点
- ADMIN: 设置权限

*(For details, go to* http://zookeeper.apache.org/doc/r3.2.2/zookeeperProgrammers.html*)*

## 4.3 样例: ZooKeeper Standalone Operation

单机模式易于启动。你需要先找到 ZooKeeper 的安装目录。

/usr/lib/zookeeper

### 4.3.1 样例代码

#### 关键类

N/A

#### 关键方法

N/A

#### 源代码

N/A

### 4.3.2 使用

本例中，你可以启动 ZooKeeper 并尝试一些基本操作。

#### 运行样例

1) 前往路径地址/usr/lib/zookeeper

2) 你需要拥有一个配置文件，才能启动 ZooKeeper。

   /usr/lib/zookeeper/conf/zoo.cfg：

   一些属性解释如下

```
tickTime = 2000 //the basic time unit in milliseconds used by
```

```
dataDir = /var/zookeeper //the location to store the in-memory database
                         snapshots and, unless specified otherwise, the
                         transaction log of updates to the database

clientPort = 2181 //the port to listen for client connections
```

3) 现在，你可以使用如下指令启动 ZooKeeper：

```
bin/zkServer.sh start
```

4) 当你启动了 ZooKeeper 后，你可以通过如下指令连接 ZooKeeper：

```
bin/zkCli.sh 127.0.0.1:2181
```

注意：如果你想要启动复制的 ZooKeeper，你只需要编辑 conf/zoo.cfg 文件，例如：

```
tickTime=2000
dataDir=/var/zookeeper
clientPort=2181
initLimit=5  //the timeouts ZooKeeper uses to limit the length of time the
               ZooKeeper servers in quorum have to connect to a leader
syncLimit=2   //how far out of data a server can be from a leader
server.1=zoo1:2888:3888   //first port: connect to other peers;second port:
                            leader election
server.2=zoo2:2888:3888
server.3=zoo3:2888:3888
```

5) 你可以输入指令*–help* 得到 ZooKeeper 中指令使用方法。

### 分析结果

你应得到的 ZooKeeper 中的帮助界面如下：

```
[zk: 127.0.0.1:2181(CONNECTING) 1] -help
ZooKeeper -server host:port cmd args
          connect host:port
          get path [watch]
          ls path [watch]
          set path data [version]
          delquota [-n|-b] path
          quit
          printwatches on|off
          create [-s] [-e] path data acl
          stat path [watch]
          close
          ls2 path [watch]
```

## 4.4 样例： ZooKeeper API

***ZooKeeper***

ZooKeeper 通常被用来使用其 jar 包创建实例并调用其接口。主要操作为添加删除 znode 和监控 znode 的变化。

### 4.4.1 样例代码

#### 关键类

N/A

#### 关键方法

N/A

#### 源代码

如下为 ZooKeeper 提供的主要 api：

```
//create a ZooKeeper instance, the first parameter is the goal server's address
and
Port, the second is the timeout of Session, and the third is the call back method
when the node changes
ZooKeeper zk = new ZooKeeper("127.0.0.1:2181", 500000,new Watcher() {
            //monitor all the triggered events
              public void process(WatchedEvent event) {
            //dosomething
            }
      });

//create a root node with the data of mydata, no ACL control permission, and the
node is eternal(the node will not disappear even when the client is shutdown)
zk.create("/root", "mydata".getBytes(),Ids.OPEN_ACL_UNSAFE, CreateMode.PERSI
STENT);

//create a clildone znode with the data of childone below root
zk.create("/root/childone","childone".getBytes(), Ids.OPEN_ACL_UNSAFE,Create
Mode.PERSISTENT);

//get the names of children nodes of root, return List<String>
 zk.getChildren("/root",true);

//get the data of /root/childone, return byte[]
 zk.getData("/root/childone", true, null);

//edit the data of /root/childone
 zk.setData("/root/childone","childonemodify".getBytes(), -1);
```

```
//delete the /root/childone node, the second parameter is version, if it is set to be -1, then it is deleted directly.
  zk.delete("/root/childone", -1);
//close the session
zk.close();
```

### 4.4.2 使用

本样例提供 ZooKeeper api 的一些基本使用。

#### 运行样例

N/A

#### 分析结果

N/A

# 5 Hive

## 5.1 前提

确定 Hadoop，HBase，Hive 已经被正确安装，配置，并且正在运行。

## 5.2 概要

Hive 为一基于 Hadoop 的大数据分布式数据仓库软件。它可以将数据存放在分布式文件系统或分布式数据库中，并提供大数据统计、查询和分析操作。它使用 MapReduce 来执行操作，HDFS 来储存数据。

Hive 由 metastore 和 Hive 引擎组成。Metastore 负责存储架构信息并提供存储数据的一个结构体。查询处理，编译，优化，执行均由 Hive 引擎完成。

为了读取/写入数据到表格中，Hive 使用 SerDe。SerDe 提供将数据转化为存储格式，并且从一系列字节中提取数据结构的方法。

如下为 Hive 中的 shell 命令：

| 命令 | 描述 |
|---|---|
| quit | Uses quit or exit to come out of interactive shell. |
| reset | Resets configuration to the default values (as of Hive 0.10) |
| set <key>=<value> | Use this to set value of particular configuration variable. One thing to note here is that if you misspell the variable name, cli will not show an error. |
| set | This will print list of configuration variables that overridden by user or hive. |
| set -v | This will give all possible hadoop/hive configuration variables. |
| add FILE <value> <value>* | Adds a file to the list of resources. |
| list FILE | list all the resources already added |
| list FILE <value>* | Check given resources are already added or not. |
| ! <cmd> | execute a shell command from hive shell |
| dfs <dfs command> | execute dfs command from hive shell |
| <query string> | executes hive query and prints results to stdout |

*(Reference from: https://cwiki.apache.org/confluence/display/Hive/GettingStarted)*

## 5.3 样例: Count(*)

通过在 Hive 中创建一个表格，你可以执行插入，统计行数等操作。这个样例展示了使用 Hive 的一些基本操作。

### 5.3.1 样例代码

#### 关键类

N/A

#### 源代码

你可以通过 create_table.q 和 gendata.sh 文件完成该样例。

#### 关键方法

| 方法名称 | 描述 |
| --- | --- |
| Create_table_if_not_exists_ t01_party_card_rela_h() | 创建名称为 t01_party_card_rela_h 的表格 |
| gendata() | 向表格中插入数据 |

#### 源代码

create_table.q:

```
drop table if exists t01_party_card_rela_h;
CREATE TABLE IF NOT EXISTS t01_party_card_rela_h (
//define the columns of the table
  party_id string,
  card_no string,
  party_card_rela_type_cd string,
  start_date string,
  end_date string ,
  etl_job string ,
  etl_src string,
  source_system_cust_num string
) ROW FORMAT DELIMITED FIELDS TERMINATED BY '|' STORED AS TEXTFILE;
LOAD DATA LOCAL INPATH 'data_factors.dat' INTO TABLE t01_party_card_rela_h;
//load the local data into the table
```

data_factors.dat

```
1|1|normal|20121127|20121129|job1|src1|1
2|2|normal|20121128|20121130|job2|src2|2
3|3|normal|20121129|20121130|job3|src3|3
```

gendata.sh:

```
gen_data() {

  echo "Generate triple data for  T01_PARTY_CARD_RELA_H"

  for i in seq $1 $2;
  do
    echo "Char index: $i ..."
      hive -e "INSERT overwrite  TABLE t01_party_card_rela_h select t.* from (
             select party_id,
             concat(substr(card_no,1, $i-1),
              case when substr(card_no, $i,1)='1' then 'a'
              when substr(card_no, $i,1)='2' then 'b'
              when substr(card_no, $i,1)='3' then 'c'
              when substr(card_no, $i,1)='4' then 'd'
              when substr(card_no, $i,1)='5' then 'e'
              when substr(card_no, $i,1)='6' then 'f'
              when substr(card_no, $i,1)='7' then 'g'
              when substr(card_no, $i,1)='8' then 'h'
              when substr(card_no, $i,1)='9' then 'i'
              when substr(card_no, $i,1)='0' then 'j'
              else substr(card_no, $i,1)
              end
             ,substr(card_no, $i+1)) as card_no,
             party_card_rela_type_cd ,
             start_date ,
             end_date  ,
             etl_job  ,
             etl_src ,
             source_system_cust_num   from t01_party_card_rela_h

             UNION ALL
             select party_id,
             concat(substr(card_no,1, $i-1),
              case when substr(card_no, $i,1)='1' then 'z'
              when substr(card_no, $i,1)='2' then 'y'
              when substr(card_no, $i,1)='3' then 'x'
              when substr(card_no, $i,1)='4' then 'w'
              when substr(card_no, $i,1)='5' then 'v'
              when substr(card_no, $i,1)='6' then 'u'
              when substr(card_no, $i,1)='7' then 't'
              when substr(card_no, $i,1)='8' then 's'
              when substr(card_no, $i,1)='9' then 'r'
              when substr(card_no, $i,1)='0' then 'q'
```

***Hive***

```
             else substr(card_no, $i,1)
             end
            ,substr(card_no, $i+1)) as card_no,
            party_card_rela_type_cd ,
            start_date ,
            end_date  ,
            etl_job  ,
            etl_src ,
            source_system_cust_num    from t01_party_card_rela_h

            UNION ALL
            select party_id,
            card_no,
            party_card_rela_type_cd ,
            start_date ,
            end_date  ,
            etl_job  ,
            etl_src ,
            source_system_cust_num    from t01_party_card_rela_h
            ) t"
  done
}

gen_data 1 9
//you can change the number of insertions in the table
```

### 5.3.2 使用

你可以在 Hive 中创建新的表格，插入数据并计算行数。

#### 运行样例

1) 导入包含代码 *create_table.q* 和 *gendata.sh* 的文件夹。

2) 在 data_factors.dat 文件中创建如下数据集

   1|1|normal|20121127|20121129|job1|src1|1

   2|2|normal|20121128|20121130|job2|src2|2

   3|3|normal|20121129|20121130|job3|src3|3

3) 修改 *create_table.q* 文件中的表格名称，例如 t01_party_card_rela_h。然后运行指令：

   ```
   hive -f create_table.q
   ```

4) 进入 Hive shell：

```
hive
```

5) 列出所有表格名称：

```
show tables;
```

通过如下指令检查表格格式：

```
DESCRIBE t01_party_card_rela_h;
```

*6)* 退出 Hive shell，并修改 gendata.sh 文件中的第 71 行来确定你想插入的记录行数，然后运行指令：

```
sudo sh gendata.sh
```

7) 现在你可以查询 t01_party_card_rela_h 表格中的行数，通过在 Hive shell 中运行如下指令：

```
SELECT count(*) FROM t01_party_card_rela_h;
```

### 分析结果

如下为统计表格中行数输出的结果：

```
MapReduce Total cumulative CPU time: 5 seconds 220 msec
Ended Job = job_201208151826_0019
MapReduce Jobs Launched:
Job 0: Map: 2  Reduce: 1   Cumulative CPU: 5.22 sec   HDFS Read: 89991138 HDFS Write:
7 SUCCESS
Total MapReduce CPU Time Spent: 5 seconds 220 msec
OK
787320
Time taken: 27.633 seconds
```

你可以看到程序遍历表格所用的时间。

## 5.4 样例: Mixed Load Stress on Hive

通过运行指令同时修改同一表格，你可以检测 Hive 的可靠性。

### 5.4.1 样例代码

#### 关键类

***Hive***

与上一样例相同。

### 关键方法

| 方法名称 | 描述 |
| --- | --- |
| Create_table_if_not_exists_ t01_party_card_rela_h() | 创建名称为 t01_party_card_rela_h 的表格 |

### 源代码

Create_table.q (与上一样例相同)

## 5.4.2 使用

本样例测试了集群中 Hive 的可靠性，并带你熟悉 Hive 中的基本指令。

### 运行样例

1) 在前一样例中，你创建了表格 t01_party_card_rela_h。现在，你可以在 create_table.q 中改变表格名称为 t01_party_card_rela_h1，并执行指令：

```
hive -f create_table.q
```

2) 进入 Hive shell 并将 t01_party_card_rela_h 表格中数据插入 t01_party_card_rela_h1 表格：

```
INSERT INTO TABLE t01_party_card_rela_h1 SELECT * FROM t01_party_card_rela_h;
```

当插入程序正在执行时，复制当前集群的 session，并进入 Hive shell，运行指令：

```
DROP TABLE t01_party_card_rela_h1;
```

3) 在原先 session 中，由于 t01_party_card_rela_h1 表格的丢失，MapReduce 操作已被停止

4) 通过指令检查 HDFS 的*/user/hive/warehouse* 目录：

```
sudo -u hdfs hadoop fs -ls /user/hvie/warehouse
```

### 分析结果

你可在 */user/hive/warehouse* 目录下看到如下结果：

```
Found 1 items
drwxr-xr-x   - root hadoop          0 2012-08-16 14:11
/user/hive/warehouse/t01_party_card_rela_h
```

## 5.5 样例: DDL operation on Hive

***Hive***

该样例展示了 Hive 的一些基本操作，比如创建，删除和修改表格。

### 5.5.1 样例代码

#### 关键类

N/A

#### 关键方法

N/A

#### 源代码

该样例通过几条简单指令展示了 Hive 的一些基本功能。

### 5.5.2 使用

你可以掌握在 Hive 中创建分区表，修改表属性等操作。

#### 运行样例

1) 进入 Hive shell 并运行指令：

```
CREATE TABLE table_name(id int, dtDontQuery string, name string)
PARTITIONED BY (dt string);
```

现在你已创建有两个分区的测试表格。

2) 通过如下指令创建两个分区表：

```
ALTER TABLE table_name ADD PARTITION (dt='1');

ALTER TABLE table_name ADD PARTITION (dt='2');
```

3) 通过如下指令检查 *table_name* 表格的分区：

```
SHOW PARTITIONS table_name;
```

4) 现在，你可以选择

- 删除分区*'dt=1'*：

```
ALTER TABLE table_name DROP PARTITION (dt='1');
```

- 检查 *table_name* 表格中的数据格式：

```
DESCRIBE table_name;
```

- 更改 *table_name* 表格中的数据名称：

```
ALTER TABLE table_name CHANGE dtdontquery dquery string;
```

- 为表格添加新的列：

```
ALTER TABLE table_name ADD COLUMNS (new1 string, new2 int);
```

- 将表格 table_name 重命名为 new_table_name：

```
ALTER TABLE table_name RENAME TO new_table_name;
```

### 分析结果

如果你修改了 *table_name* 表格中数据名称并按照样例步骤添加了新的数据，执行下列命令，你会得到如下结果：

```
DESCRIBE new_table_name;

hive> describe new_table_name;
OK
id      int
dquery  string
name    string
new1    string
new2    int
dt      string
Time taken: 0.033 seconds
```

## 5.6 样例：Hive over HBase

Hive over hbase 是基于 hive 支持对 hbase 表提供直接查询功能。

### 查询语句支持

字段类型: boolean, tinyint, smallint, int, bigint, float, double, string, struct

(当 hbase 中的 rowkey 字段为 struct 类型，请将子字段定义为 string 类型，同时指定表的 collection items terminated 分隔字符以及各子字段的长度参数:hbase.rowkey.column.length)

Where 子句

Groupby，having 子句

聚合函数: count, max, min, sum, avg

Orderby with limit 子句 (top N)

limit 子句

explain

关系操作：>, >=, <=, <, =

算术操作：+, -, *, /, %

逻辑操作：and, or, not

字符串操作函数: substring, concat

Distinct : 支持 select distinct <col-list> from <tab> where <expr>和 select aggr-fun(distinct <col_list>) from <tab> where <expr>

Like: 通配符为'_', '%'

## 不支持

Sub-query

Join

Union

### 5.6.1 样例代码

**关键类**

N/A

**源代码**

你可以通过 init_table.q 文件完成该样例。

init_table.q

```
CREATE TABLE pokes (foo INT, bar STRING);
LOAD DATA LOCAL INPATH './data/files/kv1.txt' OVERWRITE INTO TABLE pokes;

DROP TABLE t_hbase;
CREATE TABLE t_hbase(key STRING,
                  tinyint_col TINYINT,
                  smallint_col SMALLINT,
                  int_col INT,
                  bigint_col BIGINT,
                  float_col FLOAT,
                  double_col DOUBLE,
                  boolean_col BOOLEAN)
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH           SERDEPROPERTIES           ("hbase.columns.mapping"           =
":key#-,cf:binarybyte#-,cf:binaryshort#-,cf:binaryint#-,cf:binarylong#-,cf:b
inaryfloat#-,cf:binarydouble#-,cf:binaryboolean#-")
TBLPROPERTIES ("hbase.table.name" = "t_hive",
```

```
            "hbase.table.default.storage.type" = "binary");

INSERT OVERWRITE TABLE t_hbase
SELECT bar as key, 1, 11, foo, cast(foo as bigint) bigint_col, 1.0, 1.0, true
FROM pokes
WHERE foo>100 and foo<200;

INSERT OVERWRITE TABLE t_hbase
SELECT bar as key, 1, 22, foo, cast(foo as bigint) bigint_col, 1.0, 2.0, false
FROM pokes
WHERE foo>200 and foo<400;

INSERT OVERWRITE TABLE t_hbase
SELECT bar as key, 1, 33, foo, cast(foo as bigint) bigint_col, 1.0, 3.0, false
FROM pokes
WHERE foo>400 and foo<500;

INSERT OVERWRITE TABLE t_hbase
SELECT bar as key, 1, 44, foo, cast(foo as bigint) bigint_col, 1.0, 4.0, false
FROM pokes
WHERE foo>500 and foo<700;

DROP TABLE t_hbase_2;
CREATE TABLE t_hbase_2(key struct<col1:string, col2:string, col3:string>,
                  tinyint_col TINYINT,
                  smallint_col SMALLINT,
                  int_col INT,
                  bigint_col BIGINT,
                  float_col FLOAT,
                  double_col DOUBLE,
                  boolean_col BOOLEAN)
row format delimited
 collection items terminated by '|'
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH           SERDEPROPERTIES           ("hbase.columns.mapping"           =
":key#-,cf:binarybyte#-,cf:binaryshort#-,cf:binaryint#-,cf:binarylong#-,cf:b
inaryfloat#-,cf:binarydouble#-,cf:binaryboolean#-")
TBLPROPERTIES ("hbase.table.name" = "t_hive_2",
            "hbase.rowkey.column.length"="7,3,4"
);

INSERT OVERWRITE TABLE t_hbase_2
SELECT struct(bar,cast(foo as string),'aaaa') as key, 1, 11, foo, cast(foo as
bigint) bigint_col, 1.0, 1.0, true
```

***Hive***

```
FROM pokes
WHERE foo>100 and foo<300;

INSERT OVERWRITE TABLE t_hbase_2
SELECT struct(bar,cast(foo as string),'aaaa') as key, 1, 22, foo, cast(foo as
bigint) bigint_col, 1.0, 2.0, false
FROM pokes
WHERE foo>300 and foo<500;
```

### 5.6.2 使用

#### 运行样例

1） 导入包含代码 init_table.q 和'./data/files/kv1.txt'的文件夹。（./data/files/kv1.txt 是开源项目 apache hive 的测试样例文件 http://hive.apache.org/releases.html#Download）

2） 修改 init_table.q 文件中的表格名称，然后运行指令：

```
hive -f init_table.q
```

3） 进入 Hive shell：

```
hive
```

4） 执行 hive 命令，执行查询语句。

先设置以下 conf 参数

```
set hive.exec.storagehandler.local=true;
```

即可以使用 hive over hbase 的功能。

```
select count(0) from t_hbase;
```

```
hive> select count(0) from t_hbase;
2012-11-29 02:16:01     Launching Stage-2
Hbase ExecGroupby start
2012-11-29 02:16:01     End of HbaseQuery task; Time Taken: 0.025 sec.
result size==1
2012-11-29 02:16:01     Stage-2 Hbase ExecGroupby completed successfully
OK
249
Time taken: 0.117 seconds
```

```
select * from t_hbase where key='val_201' limit 5;
```

```
hive> select * from t_hbase where key='val_201' limit 5;
get One Column RowKeyRanges start
2012-11-29 02:15:48     Launching Stage-2
Hbase execFilter start
2012-11-29 02:15:48     End of HbaseQuery task; Time Taken: 0.012 sec.
2012-11-29 02:15:48     Stage-2 Hbase execFilter completed successfully
OK
val_201 1       22      201     201     1.0     2.0     false
Time taken: 0.097 seconds
```

```
select * from t_hbase where key>'val_201' and key<'val_301' and bigint_col=209
limit 5;
```

```
hive> select * from t_hbase where key>'val_201' and key<'val_301' and bigint_col
=209 limit 5;
get One Column RowKeyRanges start
2012-11-29 02:15:32     Launching Stage-2
Hbase execFilter start
2012-11-29 02:15:32     End of HbaseQuery task; Time Taken: 0.019 sec.
2012-11-29 02:15:32     Stage-2 Hbase execFilter completed successfully
OK
val_209 1       22      209     209     1.0     2.0     false
Time taken: 0.118 seconds
```

```
select smallint_col, count(tinyint_col) as cnt from t_hbase group by
smallint_col having cnt>100;
```

```
hive> select smallint_col, count(tinyint_col) as cnt from t_hbase group by small
int_col having cnt>100;
2012-11-29 02:15:14     Launching Stage-2
Hbase ExecGroupby start
2012-11-29 02:15:14     End of HbaseQuery task; Time Taken: 0.023 sec.
result size==1
2012-11-29 02:15:14     Stage-2 Hbase ExecGroupby completed successfully
OK
22      119
Time taken: 0.113 seconds
```

```
select * from t_hbase order by int_col limit 5;
```

```
hive> select * from t_hbase order by int_col limit 5;
2012-11-29 02:14:57     Launching Stage-2
Hbase execFilter start
2012-11-29 02:14:57     End of HbaseQuery task; Time Taken: 0.044 sec.
result size==5
2012-11-29 02:14:57     End of HbaseQuery task; Time Taken: 0.045 sec.
2012-11-29 02:14:57     Stage-2 Hbase execFilter completed successfully
OK
val_103 1       11      103     103     1.0     1.0     true
val_104 1       11      104     104     1.0     1.0     true
val_105 1       11      105     105     1.0     1.0     true
val_111 1       11      111     111     1.0     1.0     true
val_113 1       11      113     113     1.0     1.0     true
Time taken: 0.152 seconds
```

```
select distinct int_col from t_hbase;
```

```
hive> select distinct int_col from t_hbase;
2012-11-29 02:16:49     Launching Stage-2
Hbase execFilter start
select distinct::toInteger(columnValue("cf", "binaryint"))
2012-11-29 02:16:49     End of HbaseQuery task; Time Taken: 0.079 sec.
result size==249
2012-11-29 02:16:49     End of HbaseQuery task; Time Taken: 0.085 sec.
2012-11-29 02:16:49     Stage-2 Hbase execFilter completed successfully
OK
103
104
105
```

```
493
494
495
496
497
498
Time taken: 0.211 seconds
```

（限于版面，本例仅截图开始和最后部分）

```
select count(distinct int_col) from t_hbase;
```

```
hive> select count(distinct int_col) from t_hbase;
2012-11-29 02:13:08     Launching Stage-2
Hbase ExecGroupby start
2012-11-29 02:13:09     End of HbaseQuery task; Time Taken: 0.094 sec.
result size==1
2012-11-29 02:13:09     Stage-2 Hbase ExecGroupby completed successfully
OK
249
Time taken: 0.187 seconds
```

```
select * from t_hbase_2 where key.col1 = 'val_315';
```

```
hive> select * from t_hbase_2 where key.col1 = 'val_315';
2012-11-29 02:14:05     Launching Stage-2
Hbase execFilter start
2012-11-29 02:14:05     End of HbaseQuery task; Time Taken: 0.013 sec.
2012-11-29 02:14:05     Stage-2 Hbase execFilter completed successfully
OK
{"col1":"val_315","col2":"315","col3":"aaaa"}   1       22      315     315     1
.0      2.0     false
Time taken: 0.077 seconds
```

```
select * from t_hbase_2 where key.col1 like '_al\\_315' or key.col1 like
'_al\\_104';
```

```
hive> select * from t_hbase_2 where key.col1 like '_al\\_315' or key.col1 like '
_al\\_104';
2012-11-29 02:14:20     Launching Stage-2
Hbase execFilter start
2012-11-29 02:14:20     End of HbaseQuery task; Time Taken: 0.013 sec.
2012-11-29 02:14:20     Stage-2 Hbase execFilter completed successfully
OK
{"col1":"val_104","col2":"104","col3":"aaaa"}   1       11      104     104     1
.0      1.0     true
{"col1":"val_315","col2":"315","col3":"aaaa"}   1       22      315     315     1
.0      2.0     false
Time taken: 0.115 seconds
```

```
select * from t_hbase_2 where key.col1 in ("val_315","val_104","val_105") and
key.col2 like '%5';
```

```
hive> select * from t_hbase_2 where key.col1 in ("val_315","val_104","val_105")
and key.col2 like '%5';
2012-11-29 02:14:35     Launching Stage-2
Hbase execFilter start
2012-11-29 02:14:35     End of HbaseQuery task; Time Taken: 0.011 sec.
2012-11-29 02:14:35     Stage-2 Hbase execFilter completed successfully
OK
{"col1":"val_105","col2":"105","col3":"aaaa"}   1       11      105     105     1
.0      1.0     true
{"col1":"val_315","col2":"315","col3":"aaaa"}   1       22      315     315     1
.0      2.0     false
Time taken: 0.105 seconds
```

## 5.7 样例：Hive Excel integration

1） 在 Windows 上安装 excel2010、PowerPivot 及 HiveODBC driver（在 HiveODBCSetup.zip 文件中）。需要注意的是公司电脑安装的是 32 位的 excel2010，所以 PowerPivot 和 hiveodbc driver 也应选择 32 位的。

以上软件在 hdp-storage 服务器的/mnt/disk1/SMB/excel_hive 目录下，如下图所示：

```
[root@hdp-storage excel_hive]# pwd
/mnt/disk1/SMB/excel_hive
[root@hdp-storage excel_hive]# ll
86636
-rw-r--r-- 1 root root  50449456  90 12:30 dotNetFx40_Full_x86_x64.exe
-r--r--r-- 1 root root  14846860  97 17:44 hiveodbc-2.0-2.x86_64.rpm
-rw-r--r-- 1 root root   2808493  90 13:55 HiveODBCSetup.zip
-rw-r--r-- 1 root root 126930944  90 12:24 PowerPivot_for_Excel_amd64.msi
-rw-r--r-- 1 root root  98476032  90 13:05 PowerPivot_for_Excel_x86.msi
[root@hdp-storage excel_hive]#
```

由于 PowerPivot 依赖于.net 4.0，所以我把.net 的安装文件也放在该目录下。

2） 在 Hive 服务器上安装 hiveodbc 的 rpm 包

该包在 idh 安装 iso 的 idh/hadoop_related/x86_64 目录下。需要注意的是该 rpm 应该与使用者安装的 idh 版本一致，在 hdp-storage 上的 hiveodbc-2.0-2.x86_64.rpm 仅是一个示例。该包是 KangYi 制作，有关该包问题请咨询 KangYi。

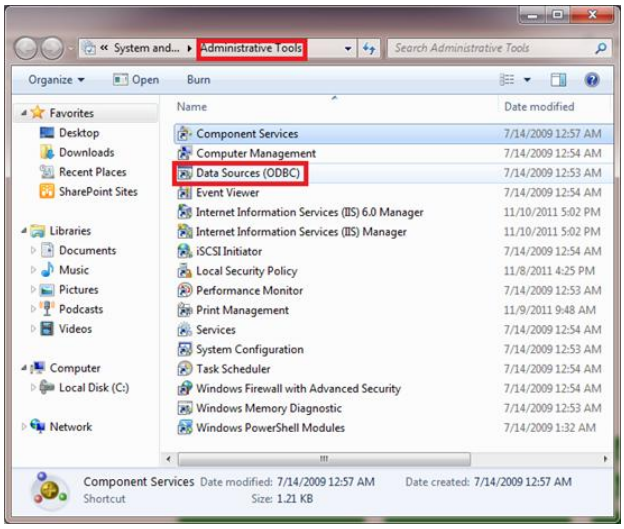
3） 安装完成该包后需要配置 hiveodbc 的配置文件/usr/etc/odbc.ini 如下所示：

```
[Hive] 【Data Source Name】
Driver = /usr/lib64/libodbchive.so.1.0.0
Description = Hive Driver v1
DATABASE = default
HOST = HIVESERVER 【Hive Server Name】
PORT = 10000
FRAMED = 0
```

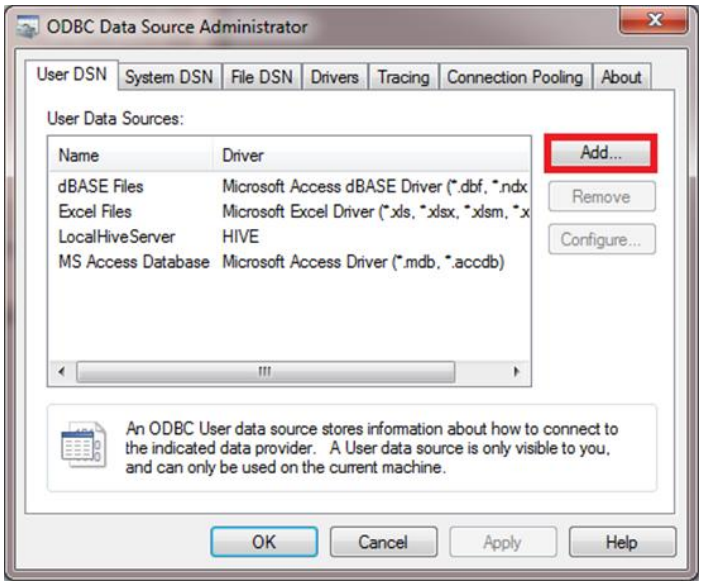
只要修改 HOST 为当前 HIVE 服务器的域名或者 IP 即可，修改该配置文件后需要重启 Hive 服务器。

4） 配置数据源

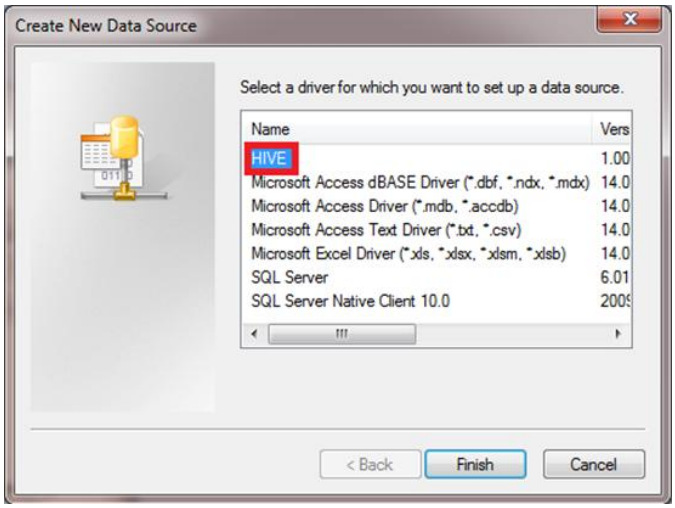
依照下图所示选择控制面板->Administrator tools -> Data Sources(ODBC)选项




选择添加数据源如下：

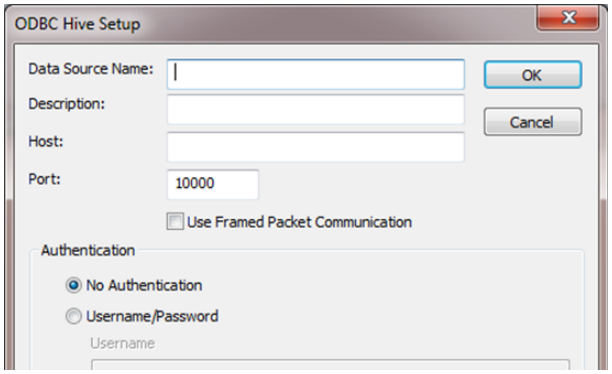



添加一个 Hive 数据源如下：

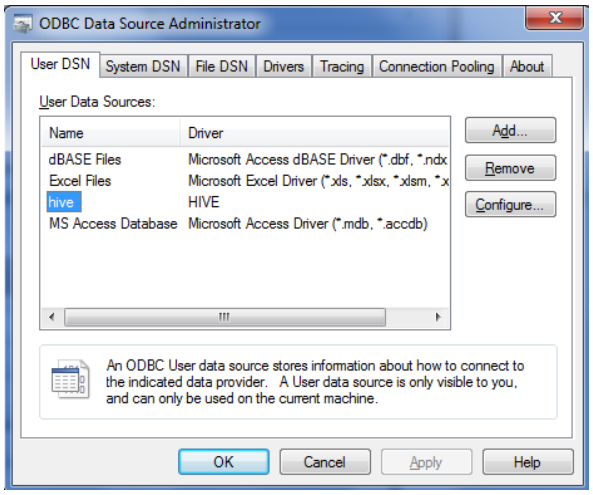



在 ODBC Hive Setup 对话框中输入相关信息如下，host 字段填 Hive 服务器主机名，port 保留默认即可，不需要验证信息：

***Hive***

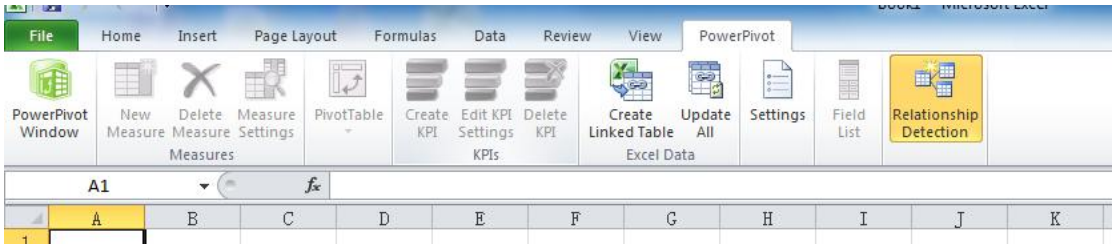



完成后如下：

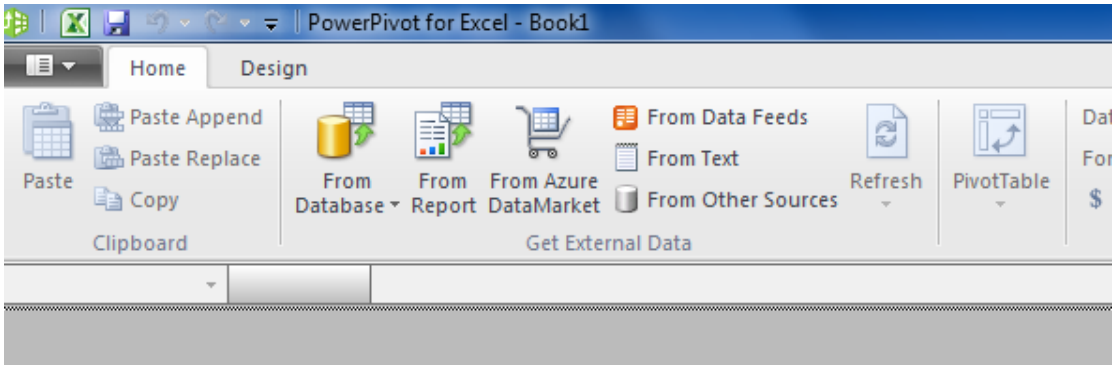



5） 使用 PowerPivot 连接 Hive

打开 excel2010，可以发现菜单面板最右侧多了 PowerPivot，如下所示：

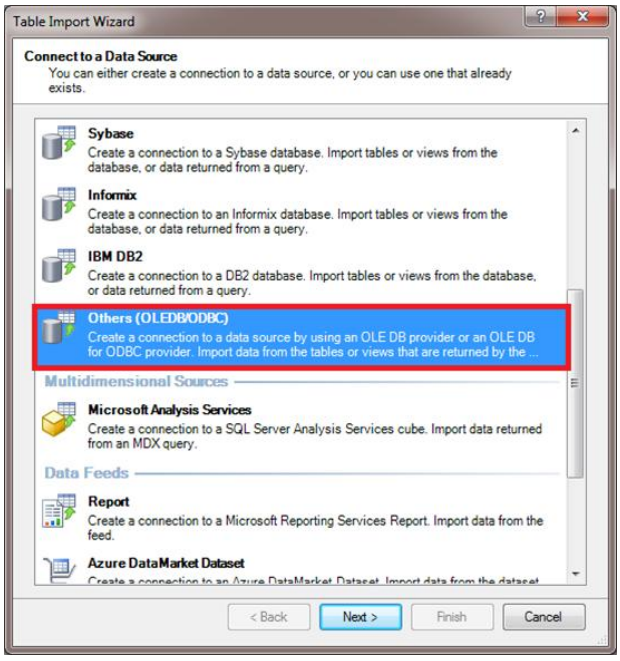



点击最左侧的 PowerPivot Window，如下图所示：

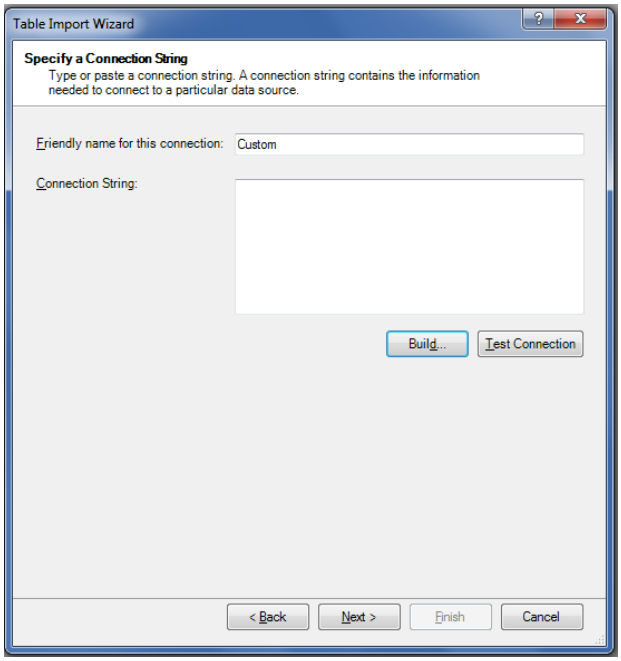



点击 From Other Sources，选择 Others(OLEDB/ODBC)

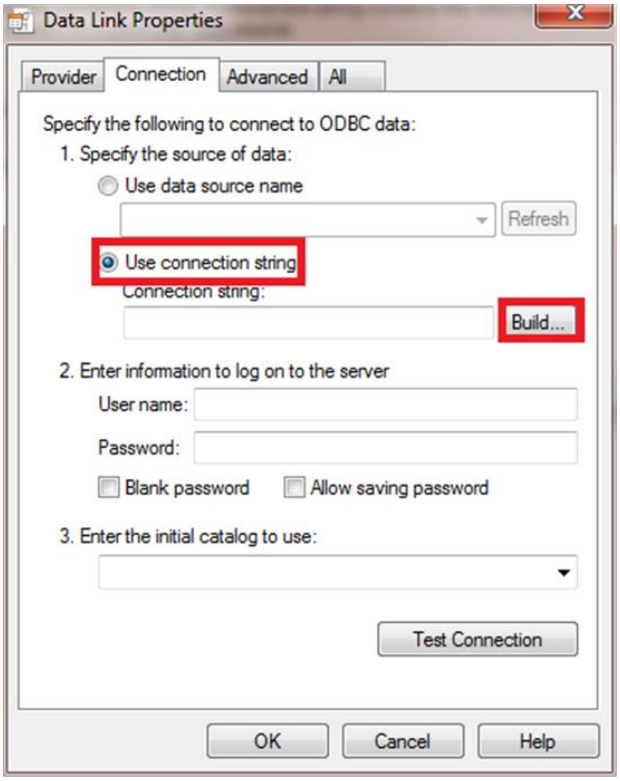
***Hive***



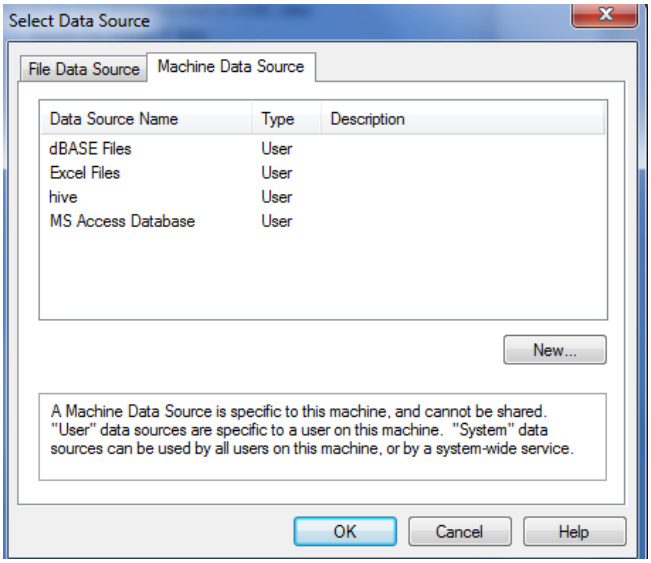

在 **table import wizard** 对话框中创建一个 Hive 连接，选择 build 选项如下所示：



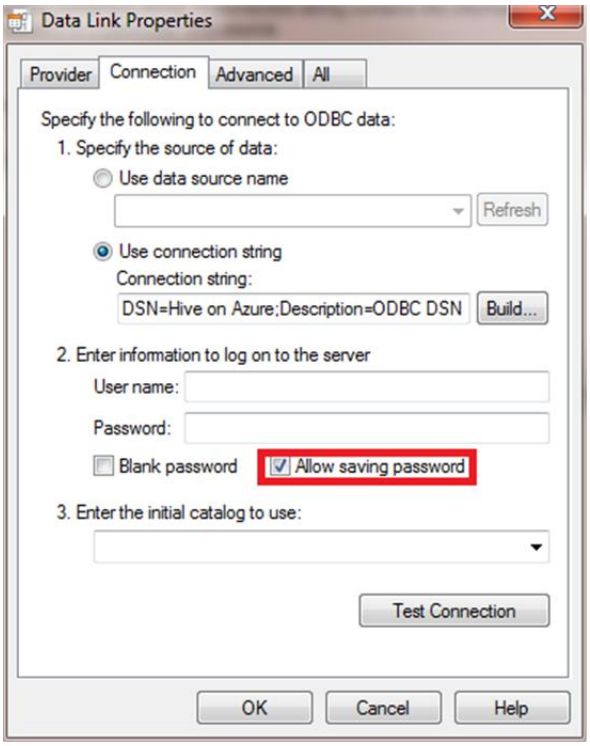

在 Data Link Properties 对话框中选择 connection 选项，选择 Use connection string 选项，并点击 build 按钮如下图所示：



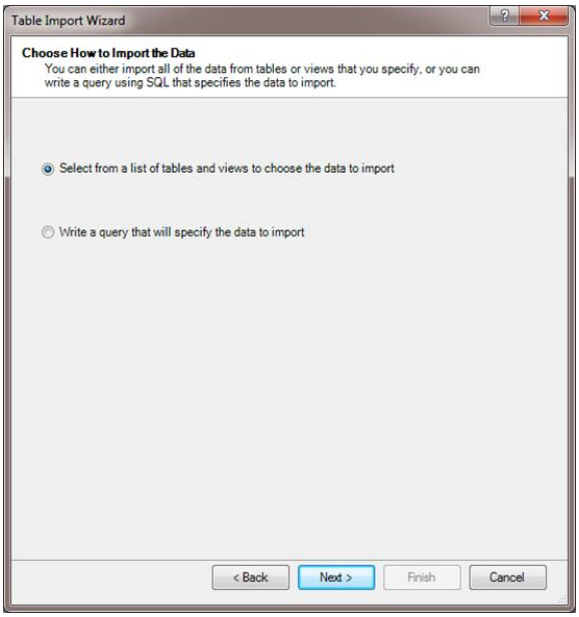

选择刚才创建的数据源，并点击 OK，如下图所示：



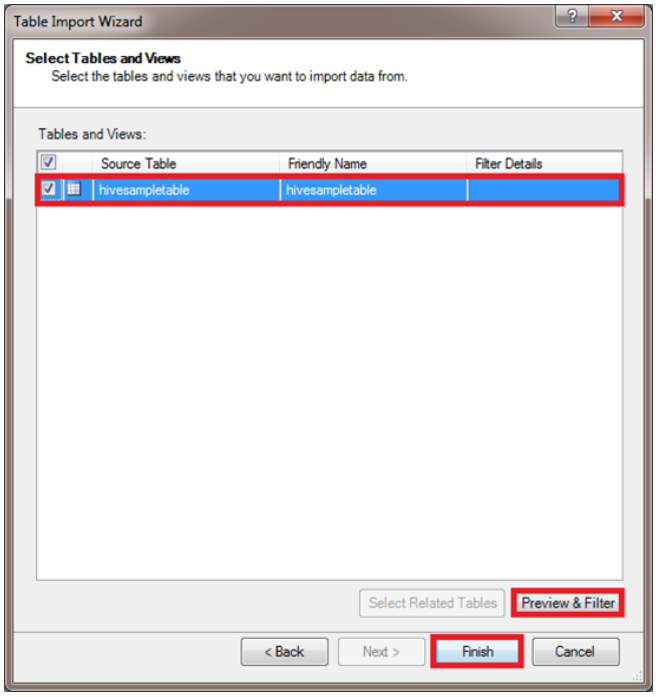

如果可以顺利连接上 Hive 服务器，即会生成 build 字符串如下图所示，同时需要勾选 **Allow Saving Password** 选项：




回到 **table import wizard** 对话框，点击 next 按钮，选择第一个选项如下图所示：



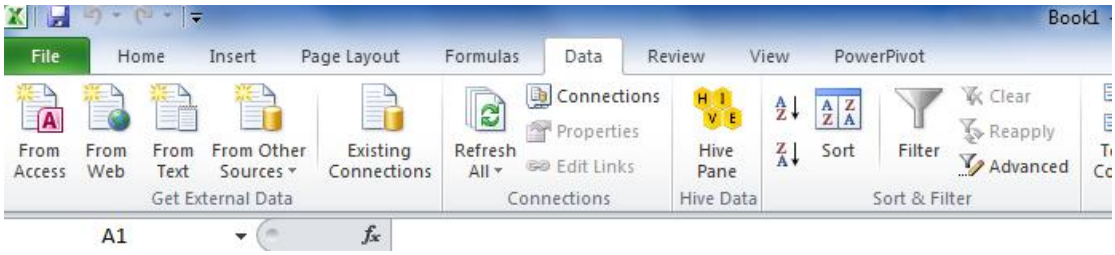

点击 next 可以看到当前 Hive Server 的所有表，如下图所示：



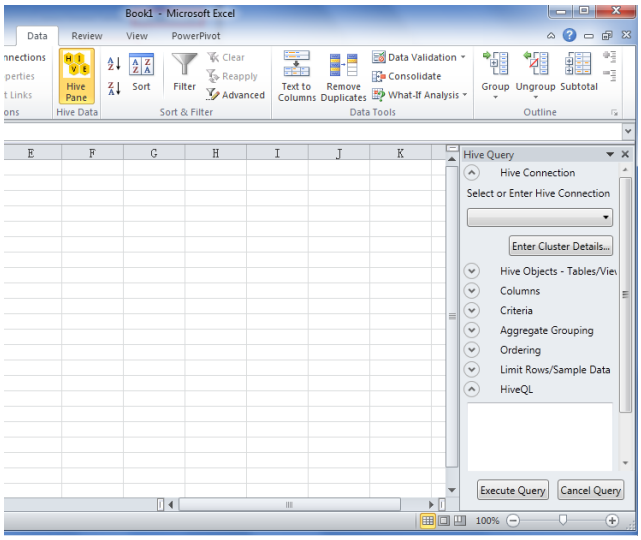

点击 Finish 完成即可。

6） Excel 整合 Hive

打开 Excel2010，在 Data 菜单面板下可以发现 Hive Pane，如下图所示：




点击后如下图所示，在左侧面板里选择 PowerPivot 中创建的 Hive 连接，然后在每个下拉框中选择相应的查询的表和字段，在最后的 HiveQL 中会生成所需的 Hive 查询语句，点击 Execute Query 即可提交 Hive 任务，在等待后台 mapreduce 完成后，会将产生的查询结果返回为一张 excel 表格。



# 6 Pig

## 6.1 前提

确定 Pig 已被正确安装，配置，并正在运行。

## 6.2 概要

作为一个海量数据的分布式数据分析语言和操作平台，Pig 的架构保证它可以分布式的并行运行任务来满足海量数据分析的需求：

Pig 的编程语言为 Pig Latin，它有如下特点：

- 易于编写

  Pig Latin 语言有 SQL 语言的灵活性和过程式语言流程性。

- 最优化策略

  Pig 系统可以自动优化执行，使得用户可以专注于语法。

- 可扩展性

  用户可以自定义功能。

有三种方法可以运行 Pig：

- Grunt Shell:

  进入 Pig shell, Grunt.

- Script File:

  运行写有 Pig 指令的脚本

- 嵌入式程序：

  将 Pig 指令嵌入程序语言并运行程序：

Pig Commands:

| 命令 | 功能 |
|---|---|
| Load | Read data from file system |
| Store | Write data to file system |
| Foreach | Apply expression to each record and output one or more records |
| Filer | Apply predicate and remove records that do not return true. |

| | |
|---|---|
| Group | Collect records with the same key from one or more inputs |
| Join | Join two or more inputs based on a key |
| Order | Sort records based on a key |
| Distinct | Remove duplicate records |
| Union | Merge two data sets |
| Split | Split data into 2 or more sets, based on filter conditions |
| Stream | Send all records through a user provided binary |
| Dump | Write output to stdout |
| Limit | Limit the number of records |

## 6.3 样例: Pig Aggregation

本样例中，你可以统计每次用户名在 excite-small.log 数据中出现的次数，然后过滤出结果，并将记录排序。

“excite-small.log”能够在 https://cwiki.apache.org/PIG/pigtutorial.data/pigtutorial.tar.gz 进行下载。

### 6.3.1 样例代码

**关键类**

N/A

**关键方法**

N/A

‘

**源代码**

```
//load the excite log file into the “raw” bag as an array of records with the fields user, time and query
log = LOAD 'excite-small.log' AS (user, timestamp, query);
//use the GROUP operator to group records by user
grpd = GROUP log BY user;
//Use the COUNT function to get the count of each group
cntd = FOREACH grpd GENERATE group, COUNT(log) AS cnt;
//filter out the result which is larger than 50
fltrd = FILTER cntd BY cnt > 50;
//order the result by cnt
srtd = ORDER fltrd BY cnt;
//Store the result into output
STORE cntd INTO 'output';
```

### 6.3.2 使用

根据本样例，你可以学习使用 Pig 指令。

#### 运行样例

通过如下指令运行该样例：

1) 进入 Pig shell：

```
pig
```

然后输入指令（不包括注释部分）。

2) 将指令输入文件 *pigtest.pig*，并在文件首部加入如下指令：

```
--pigtest.pig
```

然后运行文件：

```
pig pigtest.pig.
```

#### 分析结果

如下为/user/root/output/part-r-00000 中的部分结果：

```
FD3373744827EFA7        4
FD4BB9A09080B726        2
FD83D5C547D3EA2E        3
FDCC0A3F96D1C47A        10
FE106E193F938B17        3
FE33FDC5FAE7EB96        4
FE785BA19AAA3CBB        10
FEA681A240A74D76        4
FF5C9156B2D27FBD        1
FFA4F354D3948CFB        6
FFCA848089F3BA8C        1
```

## 6.4 样例: Pig UDF (user defined function) Sample

Pig 支持两种 UDF：eval 和 load/store。继承 eval 的用户定义功能(UDF)通常用来转换数据类型。继承 load/store 的 UDF 用来读取和存储数据类型。在本样例中，你将会实现一个将数据转化为大写字符的方法，该方法继承 eval。

### 6.4.1 样例代码

关键类

| Class | Description |
| --- | --- |
| UPPER.class | 将输入单词转化为大写 |

关键方法

| Method | Description |
| --- | --- |
| UPPER.exec() | 将数据转化为大写格式 |

源代码

UPPER.java:

```
package myudfs;
import java.io.IOException;
import org.apache.pig.EvalFunc;
import org.apache.pig.data.Tuple;
//extends the Eval class
public class UPPER extends EvalFunc<String>
{
    public String exec(Tuple input) throws IOException {
        if (input == null || input.size() == 0)
            return null;
        try{
            String str = (String)input.get(0);
            return str.toUpperCase();  //convert the string to upper case
        }catch(Exception e){
            System.out.println("Exception!");
        }
     return null;
    }

}
```

student_data

```
test1,1,1
test2,2,2
test3,3,3
```

### 6.4.2 使用

你可以依此继承 eval 或者 load/store 设计自己的功能。

## 运行样例

1) 进入 Pig 安装目录。新建一个目录叫做 myudfs。

2) 进入 myudfs 目录并将 UPPER.java 拷贝进该目录。

3) 将该目录下文件打成 jar 包：

```
javac -cp /usr/lib/pig/pig_<VERSION>-intel.jar UPPER.java

cd ..

jar -cf myudfs.jar myudfs
```

4) 在 student_data 文件中创建如下数据集：

```
test1,1,1
test2,2,2
test3,3,3
```

5) 在 Pig shell 中测试 udf：

```
pig -x local  //entering the pig shell
register myudfs.jar  //register the jar
A = load 'student_data' using PigStorage(',') as (name:chararray,
age:int,gpa:double);
B = FOREACH A GENERATE myudfs.UPPER(name);  //convert the data to upper case
dump B;  //show the result in pig
```

## 分析结果

如下为 Pig shell 输出的部分结果：

```
2012-08-23 11:23:49,091 [main] INFO
org.apache.pig.backend.hadoop.executionengine.util.MapRedUtil - Total input
paths to process : 1
(TEST1)
(TEST2)
(TEST3)
```

# 7 Mahout

## 7.1 前提

确定 Hadoop，Mahout 已经被正确安装，配置，并且正在运行。

## 7.2 概要

Mahout 为一个可扩展的机器学习库。它能和 Hadoop 结合来提供分布式数据挖掘功能，也就是说，Mahout 为基于 Hadoop 的机器学习应用程序 API。

现在，Mahout 为不同情况提供四种算法：

- 推荐引擎算法
- 聚集算法
- 分类算法
- 关联项目分析算法

## 7.3 样例: Mahout Kmeans

*K-means clustering is a method of cluster analysis which aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean.*

在 Mahout Kmeans 样例中，你可以实现一个默认运行十次迭代的 kmeans 算法。

你可以在如下地址下载该样例的代码：

`http://svn.apache.org/viewvc/mahout/tags/mahout-0.6/`

### 7.3.1 样例代码

**关键类**

| 类名 | 描述 |
|---|---|
| org.apache.mahout.clustering.syntheticcontrol.kmeans.job | 执行 kmeans 功能 |

**关键方法**

| 方法名称 | 描述 |
|---|---|
| org.apache.mahout.clustering.syntheticcontrol.kmeans.job.run() | 执行 kmeans 聚集任务 |
| org.apache.mahout.clustering.syntheticcontrol.finalClusterPath() | 返回最终迭代集群的路径 |

**源代码**

如下样例代码在 Job.class 类中：

```
/**
  * Run the kmeans clustering job on an input dataset using the given the
  * number of clusters k and iteration parameters. All output data will be
  * written to the output directory, which will be initially deleted if it
  * exists. The clustered points will reside in the path
  * <output>/clustered-points. By default, the job expects a file containing
  * equal length space delimited data that resides in a directory named
  * "testdata", and writes output to a directory named "output".
  *
  * @param conf
  *        the Configuration to use
  * @param input
  *        the String denoting the input directory path
  * @param output
  *        the String denoting the output directory path
  * @param measure
  *        the DistanceMeasure to use
  * @param k
  *        the number of clusters in Kmeans
  * @param convergenceDelta
  *        the double convergence criteria for iterations
  * @param maxIterations
  *        the int maximum number of iterations
  */
 public static void run(Configuration conf, Path input, Path output,
                    DistanceMeasure measure, int k, double convergenceDelta,
                    int maxIterations)
                    throws Exception{
   Path directoryContainingConvertedInput = new Path(output,
                                       DIRECTORY_CONTAINING_CONVERTED_INPUT);
   log.info("Preparing Input");
   InputDriver.runJob(input, directoryContainingConvertedInput,
                     "org.apache.mahout.math.RandomAccessSparseVector");
   log.info("Running random seed to get initial clusters");
   //generate random seed for the kmeans algorithm
   Path clusters = new Path(output, Cluster.INITIAL_CLUSTERS_DIR);
   clusters = RandomSeedGenerator.buildRandom(conf,
      directoryContainingConvertedInput, clusters, k, measure);
   log.info("Running KMeans");
   //run the kmeans algorithm
   KMeansDriver.run(conf, directoryContainingConvertedInput, clusters, output,
                    measure, convergenceDelta, maxIterations, true, false);
   // run ClusterDumper
   ClusterDumper clusterDumper = new ClusterDumper(finalClusterPath(conf,
```

```
                output, maxIterations), new Path(output, "clusteredPoints"));
    clusterDumper.printClusters(null);
  }
```

### 7.3.2 使用

该样例为实现 kmeans 算法的示例。

#### 运行样例

1) 导入 *synthetic_control.data* 数据集。

   (http://archive.ics.uci.edu/ml/databases/synthetic_control/synthetic_control.data）

注意：你应将数据集导入到 MAHOUT_HOME 目录下以防抛错。

2) 确保 Hadoop 已被开启。

3) 创建目录 *testdata*，并将数据集导入 *testdata*：

```
hadoop fs -mkdir testdata

hadoop fs -put <PATH TO synthetic_control.data> testdata
```

4) 运行以下命令，实现 kmeans 算法：

```
hadoop jar $MAHOUT_HOME/mahout-examples-<VERSION>-Intel-job.jar
org.apache.mahout.clustering.syntheticcontrol.kmeans.Job
```

注意：此为一行命令。

5) 当 job 完成后，你可以运行如下命令检查结果：

```
hadoop fs -lsr output

hadoop fs -get output $MAHOUT_HOME/examples

cd examples/output

ls
```

你可以看到如下结果：

```
clusteredPoints  clusters-10-final   clusters-4  clusters-7  data

clusters-0       clusters-2          clusters-5  clusters-8

clusters-1       clusters-3          clusters-6  clusters-9
```

你可以采用如下命令读取文件中的数据：

```
mahout seqdumper --seqFile /user/root/output/clusters-0/part-randomSeed
```

### 分析结果

如下为 *clusteredPoints* 文件中的内容，该文件保存了最终聚集结果。

Key-value 结果类型为(IntWritable, WeightedVecterWritable)。

```
Key: 60: Value: CL-584{n=0 c=[32.326, 31.824, 27.880, 25.925, 25.179, 28.721,
27.649, 28.002, 24.316, 34.785, 24.526, 35.989, 29.431, 25.866, 27.797, 25.021,
27.934, 30.551, 33.402, 32.927, 27.944, 26.672, 35.080, 26.011, 29.259, 22.165,
17.087, 17.803, 10.677, 18.119, 21.918, 21.310, 18.088, 15.059, 19.386, 10.854,
10.476, 13.094, 19.515, 15.567, 17.630, 10.506, 11.202, 15.251, 13.991, 11.519,
12.611, 13.306, 14.231, 11.816, 19.875, 17.645, 11.676, 10.299, 21.442, 17.222,
14.140, 17.398, 10.748, 11.687] r=[]}
```

存储在 *clusters-N* 中的内容为第 N 次聚集的结果。数据类型为(Text,Cluster)。

原始数据存储在 *data* 目录下。由于该数据格式为 vector，你可以通过 mahout vectordump 指令检查数据。

## 7.4 样例: Mahout Recommender

Mahout Recommender Sample 是用 Mahout 来实现协作过滤的方法，并能够推荐给客户推荐类似的产品。在该例中，它分析从电影评分网站得到的数据，并为客户推荐合适的电影。

你可以在如下地址下载该样例的代码：

`http://svn.apache.org/viewvc/mahout/tags/mahout-0.6/`

### 7.4.1 样例代码

#### 关键类

| 类名 | 描述 |
|---|---|
| org.apache.mahout.cf.tast.example.grouplens.GroupLensDataModel | 创建推荐引擎数据模型 |
| org.apache.mahout.cf.tast.example.grouplens.GroupLensRecommender | 提供推荐方法 |
| org.apache.mahout.cf.tast.example.grouplens.GroupLensRecommenderBuilder | 创建推荐对象 |
| org.apache.mahout.cf.tast.example.grouplens.GroupLensRecommenderEvaluatorRunner | 评估推荐对象的结果 |

#### 关键方法

| 方法名称 | 描述 |
|---|---|
| GroupLensdataModel.convertGLFile() | 通过逗号分隔符将结果格式化 |
| GroupLensdataModel.readResourceToTempFile() | 将源文件读入临时文件 |
| GroupLensRecommenderBuilder.buildRecommender() | 创建一个新的 |

| | GroupLensRecommender |
|---|---|

## 源代码

样例代码在 GroupLensdataModel 类和 GroupLensRecommenderEvaluatorRunner.class 类中:

```
  private static File convertGLFile(File originalFile) throws IOException {
    // Now translate the file; remove commas, then convert "::" delimiter to comma
    File resultFile = new File(new File(System.getProperty("java.io.tmpdir")),
"ratings.txt");
    if (resultFile.exists()) {
      resultFile.delete();
    }
    Writer writer = null;
    try {
      writer = new OutputStreamWriter(new FileOutputStream(resultFile),
Charsets.UTF_8);
      for (String line : new FileLineIterable(originalFile, false)) {
        int lastDelimiterStart = line.lastIndexOf(COLON_DELIMTER);
        if (lastDelimiterStart < 0) {
          throw new IOException("Unexpected input format on line: " + line);
        }
        String subLine = line.substring(0, lastDelimiterStart);
        String convertedLine =
COLON_DELIMITER_PATTERN.matcher(subLine).replaceAll(",");
        writer.write(convertedLine);
        writer.write('\n');
      }
    } catch (IOException ioe) {
      resultFile.delete();
      throw ioe;
    } finally {
      Closeables.closeQuietly(writer);
    }
    return resultFile;
  }

public static File readResourceToTempFile(String resourceName) throws
IOException {
    InputSupplier<? extends InputStream> inSupplier;
    try {
      URL resourceURL = Resources.getResource(GroupLensRecommender.class,
resourceName);
      inSupplier = Resources.newInputStreamSupplier(resourceURL);
    } catch (IllegalArgumentException iae) {
```

```
      File resourceFile = new File("src/main/java" + resourceName);
      inSupplier = Files.newInputStreamSupplier(resourceFile);
    }
    File tempFile = File.createTempFile("taste", null);
    tempFile.deleteOnExit();
    Files.copy(inSupplier, tempFile);
    return tempFile;
  }

public Recommender buildRecommender(DataModel dataModel) throws TasteException
{
    return new GroupLensRecommender(dataModel);
}
```

### 7.4.2 使用

该样例可用来测试 Mahout 数据过滤的准确性，你可以基于此尝试更多的数据挖掘方法。

#### 运行样例

1) 导入 ratings.dat 数据集(你可从 http://www.grouplens.org/ 上下载)

2) 确保 Hadoop 已启动。

3) 现在你可以执行 recommender：

```
hadoop jar $MAHOUT_HOME/mahout-examples-<VERSION>-Intel-job.jar

org.apache.mahout.cf.taste.example.grouplens.GroupLensRecommenderEvaluato
rRunner -i <installed_dir>/ratings.dat
```

注意：这是一行指令。

注意：如果 ratings。dat 文件过大，可以使用 linux 下的 split 命令将其分割成较少的行数。

4) 结果文件 *ratings.txt* 在目录*/tmp/ratings.txt* 下。

#### 分析结果

下面为 ratings.txt 中部分结果：

```
1,1193,5
1,661,3
1,914,3
1,3408,4
1,2355,5
1,1197,3
```

```
1,1287,5
1,2804,5
1,594,4
1,919,4
1,595,5
1,938,4
```

第一个数字表示用户的 ID。

第二个数据表示电影的序列号。

第三个数字为用户的评分。

# 8 Flume

## 8.1 前提

确定 Flume 已经被正确安装，配置，并且正在运行。

## 8.2 概要

Flume 提供了收集，聚集和转移海量日志文件的连接器。同时，Flume 使用 ZooKeeper 来确保配置的一致性和可用性。

Flume 具有如下特点：

- 高可靠性

  Flume 提供了端到端可靠传输和本地数据存储选项。

- 可管理性

  通过 ZooKeeper 保证配置的可用性并使用多个 master 来管理所有节点。

- 高扩展性

  用户可用 java 语言实现新的方法。

Flume NG 1.x 由如下主要组件组成；

- Event

  Event 为可被 Flume NG 传输的单位数据。

- Source

  Flume NG 接收数据的来源。

- Sink

  Sink 对应于每个 source，它是 Flume NG 的结果路径。

- Channel

  Channel 为 source 和 sink 之间的通道。

- Source and Sink Runners

  Runner 主要负责驱动 source or sink，并对终端用户不可见。

- Agent

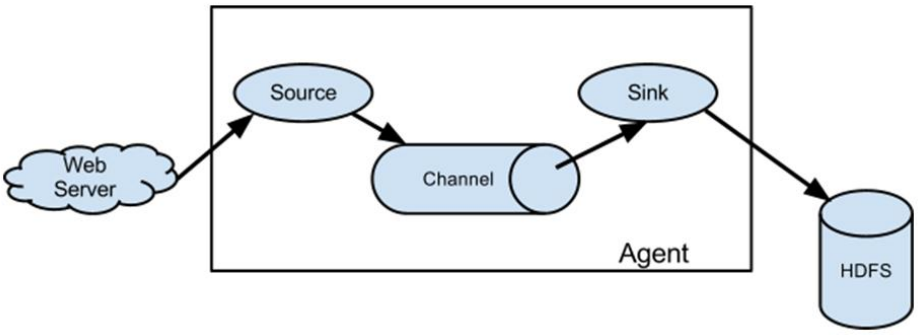
  Agent 是 Flume NG 上运行的任何物理 JVM。

- Configuration Provider

  Flume NG 有一个可插入的配置系统叫做 *configuration provide*。

  *(更多细节详见: https://cwiki.apache.org/FLUME/flume-ng.html)*

  如下，展示了 Flume 的抽象架构：




*(from:https://people.apache.org/~mpercy/flume/flume-1.2.0-incubating-SNAPSHOT/docs/FlumeUserGuide.html#data-ingestion)*

参考指令语法:

Flume-ng global options:

| **Option** | **Description** |
|---|---|
| --conf,-c <conf> | Use configs in /conf directory |
| --classpath,-C <cp> | Append to classpath |

Flum-ng node options:

| **Options** | **Description** |
|---|---|
| --conf-file,-f | Specify a config file |
| --name,-n | The name ofthis node |
| --help,-h | Display help text |

Flume-ng avro-client options:

| **Options** | **Description** |
|---|---|
| --host,-H <host> | Hostname to which events will be sent |
| --port,-p <port> | Port of the avro source |
| --filename,-F <file> | Text file to stream to avro source |
| --help,-h | Display help text |

## 8.3 样例: Flume Log Transfer

在本例中，你可以定义数据流并配置单个组件。通过启动一个 Flume 节点来启动已配置的 source 和 sink。

### 8.3.1 样例代码

#### 关键类

N/A

#### 关键方法

N/A

#### 源代码

如下为 *agent.properties* 的内容：

```
agent2.channels.ch1.type=MEMORY
#the command to be executed by flume
agent2.sources.s1.command = tail -f /var/log/messages
agent2.sources.s1.channels=ch1
agent2.sources.s1.type=EXEC
#the path to store the log
agent2.sinks.log-sink1.channel=ch1
agent2.sinks.log-sink1.type=HDFS
agent2.sinks.log-sink1.hdfs.path = hdfs://<hostname>:8020/tmp/data/testdata
# List the sources, sinks and channels for the agent
agent2.channels=ch1
agent2.sources=s1
agent2.sinks=log-sink1
```

<hostname>为可以访问 HDFS 的主机名或 IP

### 8.3.2 使用

在本例中，你可以在 Flume 中启动单个数据流。你也可以修改属性文件实现多个数据流。

#### 运行样例

1) 在 shell 中定义配置文件 agent.properties 在如下目录：

   /usr/lib/flume/conf

```
注意：如下为配置文件的示例格式：
#List the sources, sinks and channels for the agent
<agent>.sources = <Source>
<agent>.sinks = <Sink>
```

```
<agent>.channels = <Channel1> <Channel2>

#set channel for source
<agent>.sources.<Source>.channels = <Channel1> <Channel2> ...

#set channel for sink
<agent>.sinks.<Sink>.channel = <Channel1>
```

2) 启动 Flume agent：

```
bin/flume-ng node -c usr/lib/flume/conf/ -f
/usr/lib/flume/conf/agent.properties -n agent2
```

注意：该节点名称由 -n agent2 指明并且必须和 agent.properties 文件中的相符。

3) 在另一个窗口中运行如下指令添加日志：

```
logger “hello“
```

## 分析结果

在 HDFS 浏览器中检查指定路径下的日志中最近生成的文件中是否包含“hello”内容。


文件信息
文件位置: /tmp/data/testdata/FlumeData.2
内容
区块
总共页面数量:1 当前页面位置:1
SEQ!org.apache.hadoop.io.LongWritable"org.apache.hadoop.io.BytesWritable
root: hello
root: hello
root: hello

# 9 Sqoop

## 9.1 前提

确定 Sqoop，Hadoop 已经被正确安装，配置，并且正在运行。

## 9.2 概要

Sqoop 提供了在 Hadoop 和结构化数据源(例如关系型数据库)的传输模块。通过使用 Sqoop，在 MySQL, ORACLE 等外部数据库中的数据可被导入 HDFS，Hive 以及其他相关系统，也可将本地数据导出到关系型数据库和数据仓库。

如下 Sqoop 中有约 13 种指令：

| Command | Class | Function |
|---|---|---|
| Import | ImportTool | 将数据从关系型数据库导入到 HDFS |
| Export | ExportTool | 将数据从 HDFS 导出到关系型数据库 |
| Codegen | CodeGenTool | 从数据库中得到部分表格并生成对应 java 文档，并打包成 jar |
| Create-hive-table | CreateHiveTableTool | 创建一个 Hive 表格 |
| Eval | EvalSqlTool | 检查 SQL 指令的结果 |
| Import-all-tables | ImportAllTablesTool | 从一些数据库中导出所有表格到 HDFS 中 |
| Job | JobTool | |
| List-databases | ListDatabasesTool | 列出所有数据库名称 |
| List-tables | ListTableTool | 列出一些数据库中所有表格的名称 |
| Merge | MergeTool | |
| Metastore | MetastoreTool | |
| Help | HelpTool | 帮助功能 |
| Version | VersionTool | 检查版本号 |

Sqoop 中有约 10 中通用参数：

| Arguments | Description |
|---|---|
| --connect <jdbc-uri> | 指定 JDBC 连接字符串 |
| --connection-manager <class-name> | 指定使用的 connection-manager |
| --driver <class-name> | 手动指定使用的 JDBC driver |
| --hadoop-home <dir> | 覆写 $HADOOP_HOME |

| | |
|---|---|
| --help | 打印帮助指令 |
| --p | 从控制台读取密码 |
| --password <password> | 设定认证密码 |
| --username <username> | 设定认证用户名 |
| --verbose | 在运行是打印更多信息 |
| --connection-param-file <filename> | 可选的属性文件，提供更多连接参数 |

注意：你可以通过 sqoop help 指令得到指令帮助。你也可将加在任何指令之后来显示特定帮助，比如 sqoop import –help.

## 9.3 样例: Sqoop Import

Sqoop 的导入工具可将表格从 RDBMS 导入到 HDFS 中。数据是一行一行传输的，数据格式为文本文档或二进制文档。

### 9.3.1 样例代码

**关键类**

N/A

**关键方法（参数）**

| 方法名称 | 描述 |
|---|---|
| --append | 将数据加到 HDFS 上已有的一个数据集 |
| --as-textfile | 将数据导入为纯文本（默认） |
| --boundary-query <statement> | 用来创建爱你分隔符 |
| --columns <clo,col,col…> | 从表格中导出列 |
| --direct | 使用输入快捷路径 |
| --table <table-name> | 读取的表格 |
| --target-dir <dir> | HDFS 目标目录 |
| --where <where clause> | 导入时使用的 WHERE 指令 |
| --z,--compress | 允许压缩 |

**源代码**

N/A

### 9.3.2 使用

Sqoop 可以被用来从外部数据库向 HDFS, Hive, HBase 导入数据

**运行样例**

1) 连接数据库并导入数据

首先在 MySQL 中建立如下表并插入适量数据:

```
create table test_table(test_id INT primary key, test_description CHAR(32));
```

**HDFS:**

```
hadoop fs -rmr /user/root/test_table
sqoop import --connect jdbc:mysql://intelidh-01/test_db --table test_table
--username test_user -P --outdir ./hdfs_dir
```

首先清除 HDFS 中的临时文件，root 为当前用户名，test_table 为表名，下同。然后连接名为 test_db 的数据库，该数据库在 intelidh-01 主机上。你可以使用数据库主机的完整主机名或者是 IP 地址，从而将 Sqoop 和分布式 Hadoop 集群相连。

注意：Sqoop 可以连接到其他 JDBC-compliant 数据库。你需要首先导入相应 jar 包到集群目录 $SQOOP_HOME/lib 下。每个 driver.jar 包对应特定的 driver 类。例如，MySQL 的连接库含有类。你需要通过对应帮助文档找到你的数据库中主要的 driver 类，然后从 Microsoft.com 上下载对应的 driver，并将其安装在指定目录下，运行指令：

```
sqoop import -driver com.microsoft.jdbc.sqlserver.SQLServerDriver
--connect <connect-string>…
```

**Hive:**

| 参数 | 描述 |
|---|---|
| --hive-import | 将表格导入 Hive |
| --create-hive-table | 若该属性设为 true，如果目标表格在 Hive 中已存在，任务会失败。 |
| --hive-table <table-name> | 设置导入表格的名称 |
| --hive-drop-import-delims | 将表格导入 Hive 时从字符串部分去除\n,\r and \01 |

例如，你可以通过如下指令将数据导入 Hive：

```
sqoop import --connect jdbc:mysql://intelidh-01/test_db --table test_table
--username test_user -P --hive-import --hive-table test_hive_table
--outdir ./hive_dir
```

**HBase:**

| 参数 | 描述 |
|---|---|
| --column-family <family> | 为导入表格设置目标列簇 |
| --hbase-create-table | 如果已设置，会创建缺失的 HBase 表格 |
| --hbase-row-key <col> | 确定输入的列作为行关键字 |
| --hbase-table <table-name> | 确定一个 HBase 表格用作目标 |

例如，你可以通过如下指令将数据导入 HBase，导入 HBase 之前需要在 HBase 中建立相应表结构：

```
hbase(main):001:0> create 'test_hbase_table','test_column'
```

```
sqoop import --connect jdbc:mysql://intelidh-01/test_db --table test_table --username test_user -P -column-family test_column -hbase-table test_hbase_table --outdir ./hbase_dir
```

2) 你可以定义不同的参数指定表格不同的部分，将其导入 HDFS。

```
sqoop import --connect jdbc:mysql://intelidh-01/test_db --table test_table --username test_user -P --outdir ./append_dir --where "test_id > 0" --target-dir /user/root/test_table -append
```

## 分析结果

导入 HDFS:

```
12/11/29 14:20:21 INFO mapred.JobClient: Map output records=3
12/11/29 14:20:21 INFO mapred.JobClient: SPLIT_RAW_BYTES=217
12/11/29 14:20:21 INFO mapreduce.ImportJobBase: Transferred 24 bytes in 25.0308 seconds (0.9588 bytes/sec)
12/11/29 14:20:21 INFO mapreduce.ImportJobBase: Retrieved 3 records.
```

导入 Hive:

```
12/11/29 15:47:16 INFO hive.HiveImport: Logging initialized using configuration in jar:file:/usr/lib/hive/lib/hive-common-0.9.0-Intel.jar!/hive-log4j.properties
12/11/29 15:47:16 INFO hive.HiveImport: Hive history file=/tmp/root/hive_job_log_root_201211291547_1454172112.txt
12/11/29 15:47:16 INFO hive.HiveImport: OK
12/11/29 15:47:16 INFO hive.HiveImport: Time taken: 0.203 seconds
12/11/29 15:47:16 INFO hive.HiveImport: Loading data to table default.test_hive_table
12/11/29 15:47:17 INFO hive.HiveImport: OK
12/11/29 15:47:17 INFO hive.HiveImport: Time taken: 0.351 seconds
12/11/29 15:47:17 INFO hive.HiveImport: Hive import complete.
```

导入 HBase：

```
12/11/29 15:40:02 INFO mapred.JobClient: Map output records=3
12/11/29 15:40:02 INFO mapred.JobClient: SPLIT_RAW_BYTES=217
12/11/29 15:40:02 INFO mapreduce.ImportJobBase: Transferred 0 bytes in 25.2019 seconds (0 bytes/sec)
12/11/29 15:40:02 INFO mapreduce.ImportJobBase: Retrieved 3 records.
```

增量导入:

```
12/11/29 15:45:16 INFO mapreduce.ImportJobBase: Transferred 24 bytes in 25.0327
seconds (0.9587 bytes/sec)
12/11/29 15:45:16 INFO mapreduce.ImportJobBase: Retrieved 3 records.
12/11/29 15:45:16 INFO util.AppendUtils: Appending to directory test_table
12/11/29 15:45:16 INFO util.AppendUtils: Using found partition 2
```

## 9.4 样例: Sqoop Export

Sqoop 的导出工具提供了将表格从 HDFS 导出到 RDBMS 的功能。你可以定义输入文件解析的分隔符。在默认模式下，这些数据将被插入数据库，但在更新模式下，Sqoop 会生成更新指令，替换数据库当前已有的数据。

### 9.4.1 样例代码

**关键类**

N/A

**关键方法（参数）**

| 方法名称 | 描述 |
|---|---|
| --direct | 使用快速导出路径 |
| --export-dir <dir> | 将被导出的源文件路径 |
| --table <table-name> | 目标表格 |
| --update-mode <mode> | 当发现数据库中未存在的新的记录时，指定更新如何执行。合法值有 updateonly (default) 和 allowinsert |
| --input-null-string <null-string> | 对于 string 列指定将被翻译为 null 的 string 值 |

**源代码**

N/A

### 9.4.2 使用

Sqoop 可被用来将数据从 HDFS 导出到外部数据库中。

**运行样例**

你可以将数据导出到 test_table 表格中：

```
sqoop export --connect jdbc:mysql://intelidh-01/test_db --table test_table
--username test_user -P --export-dir /user/root/test_table --outdir ./export_dir
```

注意：目标表格必须已存在于数据库中。

```
12/11/29 16:01:53 INFO mapred.JobClient:    CPU time spent (ms)=1130
12/11/29 16:01:53 INFO mapred.JobClient:       Total committed heap usage
(bytes)=170000384
12/11/29 16:01:53 INFO mapred.JobClient:          Virtual memory (bytes)
snapshot=2945273856
12/11/29 16:01:53 INFO mapred.JobClient:    Map output records=3
12/11/29 16:01:53 INFO mapred.JobClient:    SPLIT_RAW_BYTES=230
12/11/29 16:01:53 INFO mapreduce.ExportJobBase: Transferred 266 bytes in 29.8986
seconds (8.8967 bytes/sec)
12/11/29 16:01:53 INFO mapreduce.ExportJobBase: Exported 3 records.
```

你也可以在数据库中查询结果。

```
mysql> select * from test_table;
+---------+------------------+
| test_id | test_description |
+---------+------------------+
|       1 | test1            |
|       3 | test3            |
|       2 | test2            |
+---------+------------------+
3 rows in set (0.00 sec)
```

# 10 常见问题

## 10.1 JoinPoint 类不存在

### 描述：

| Exception: NoClassDefFoundError: org/aspectj/lang/JoinPoint |
|---|

### 解决方法：

将/usr/lib/hadoop/lib/中的 aspectjtools-1.6.5.jar 及 aspectjrt-1.6.5.jar 加入到用户工程的 classpath 中。

# 参考


- *The Map/Reduce Tutorial*, Hadoop apache organization
- ZooKeeper: *http://zookeeper.apache.org/doc/r3.1.2/zookeeperStarted.html*
- Pig: *http://pig.apache.org/docs/r0.8.1/tutorial.html*
- Mahout: *https://cwiki.apache.org/MAHOUT/quickstart.html*
- Pig: *http://wiki.apache.org/pig/RunPig*
- Flume: *https://cwiki.apache.org/FLUME/getting-started.html*
- Hive: *https://cwiki.apache.org/confluence/display/Hive/GettingStarted*


***Reference***