Hadoop Design and *k*-Means Clustering

Kenneth Heafield

Google Inc

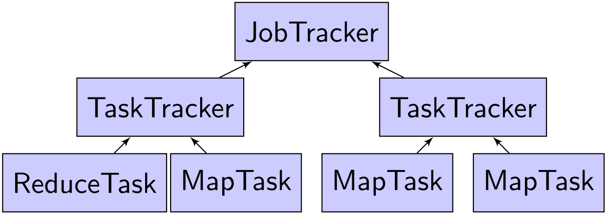
January 15, 2008

Hadoop Design

1. Fault Tolerance
2. Data Flow
   - Input
   - Output
3. MapTask
   - Map
   - Partition
4. ReduceTask
   - Fetch and Sort
   - Reduce

Later in this talk: Performance and *k*-Means Clustering

Managing Tasks



Design

- TaskTracker reports status or requests work every 10 seconds
- MapTask and ReduceTask report progress every 10 seconds

Issues

+ Detects failures and slow workers quickly
- JobTracker is a single point of failure

Coping With Failure

Failed Tasks

Rerun map and reduce as necessary.

Slow Tasks

Start a second backup instance of the same task.

Consistency

- Any MapTask or ReduceTask might be run multiple times
- Map and Reduce should be functional

Use of Random Numbers

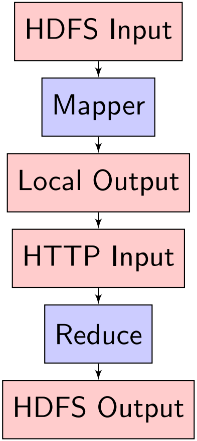
Purpose

Support randomized algorithms while remaining consistent

Sampling Mapper

```
private Random rand;
void configure(JobConf conf) {
  rand.setSeed((long)conf.getInt("mapred.task.partition"));
}
void map(WritableComparable key, Writable value,
         OutputCollector output, Reporter reporter) {
  if (rand.nextFloat() < 0.1) {
    output.collect(key, value);
  }
}
```

Data Flow



HDFS Input

Mapper

Local Output

HTTP Input

Reduce

HDFS Output

InputFormat splits and reads files

SequenceFileOutputFormat writes serialized values

Map outputs are retrieved over HTTP and merged

OutputFormat writes a SequenceFile or text

InputSplit

Purpose

Locate a single map task's input.

Important Functions

```
Path FileSplit.getPath();
```

Implementations

- `MultiFileSplit` is a list of small files to be concatenated.
- `FileSplit` is a file path, offset, and length.
- `TableSplit` is a table name, start row, and end row.

RecordReader

Purpose

Parse input specified by `InputSplit` into keys and values. Handle records on split boundaries.

Important Functions

```
boolean next(Writable key, Writable value);
```

Implementations

- `LineRecordReader` reads lines. Key is an offset, value is the text.
- `KeyValueLineRecordReader` reads delimited key-value pairs.
- `SequenceFileRecordReader` reads a `SequenceFile`, Hadoop's binary representation of key-value pairs.

InputFormat

Purpose

Specifies input file format by constructing `InputSplit` and `RecordReader`.

Important Functions

```
RecordReader getRecordReader(InputSplit split, JobConf job,
                             Reporter reporter);
InputSplit[] getSplits(JobConf job, int numSplits);
```

Implementations

- `TextInputFormat` reads text files.
- `TableInputFormat` reads from a table.

OutputFormat

Purpose

- Machine or human readable output.
- Makes `RecordWriter`, which is analogous to `RecordReader`

Important Functions

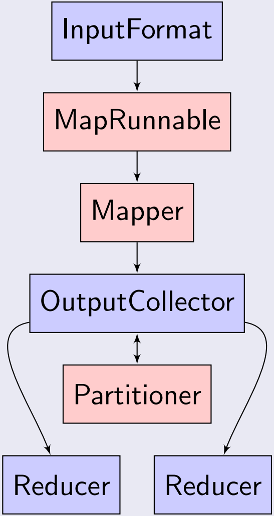
```
RecordWriter getRecordWriter(FileSystem fs, JobConf job,
  String name, Progressable progress);
```

Formats

- `SequenceFileOutputFormat` writes a binary `SequenceFile`
- `TextOutputFormat` writes text files

MapTask

Default Setup



Split files and read records

Map all records in the task

Map a record

Consult Partitioner and save files

Assign key-value pairs to reducers

Reducers retrieve files over HTTP

MapRunnable

Purpose

Sequence of map operations

Default Implementation

```
public void run(RecordReader input, OutputCollector output,
                Reporter reporter) throws IOException {
  try {
    WritableComparable key = input.createKey();
    Writable value = input.createValue();
    while (input.next(key, value)) {
      mapper.map(key, value, output, reporter);
    }
  } finally {
    mapper.close();
  }
}
```

Mapper

Purpose

Single map operation

Important Functions

```
void map(WritableComparable key, Writable value,
         OutputCollector output, Reporter reporter);
```

Pre-defined Mappers

- `IdentityMapper`
- `InverseMapper` flips key and value.
- `RegexMapper` matches regular expressions set in job.
- `TokenCountMapper` implements word count map.

Partitioner

Purpose

Decide which reducer handles map output.

Important Functions

```
int getPartition(WritableComparable key, Writable value,
                 int numReduceTasks);
```

Implementations

- HashPartitioner uses key.hashCode() % numReduceTasks.
- KeyFieldBasedPartitioner hashes only part of key.

Fetch and Sort

Fetch

- TaskTracker tells Reducer where mappers are
- Reducer requests input files from mappers via HTTP

Merge Sort

- Recursively merges 10 files at a time
- 100 MB in-memory sort buffer
- Calls key's Comparator, which defaults to key.compareTo

Important Functions

```
int WritableComparable.compareTo(Object o);
int WritableComparator.compare(WritableComparable a,
                               WritableComparable b);
```

Reduce

Important Functions

```
void reduce(WritableComparable key, Iterator values,
            OutputCollector output, Reporter reporter);
```

Pre-defined Reducers

- IdentityReducer
- LongSumReducer sums LongWritable values

Behavior

Reduce cannot start until all Mappers finish and their output is merged.

Using Hadoop

5. Performance
    - Combiners

6. *k*-Means Clustering
    - Algorithm
    - Implementation

Performance

Why We Care

- $\geq 10,000$ programs
- Average $100,000$ jobs/day
- ≥ 20 petabytes/day

Source: Dean, Jeffrey and Ghemawat, Sanjay. MapReduce: Simplified Data Processing on Large Clusters. Commun. ACM **51** (2008), 107–113.

Barriers

Concept

Barriers wait for N things to happen

Examples

- Reduce waits for all Mappers to finish
- Job waits for all Reducers to finish
- Search engine assembles pieces of results

Moral

Worry about the maximum time. This implies balance.

Combiner

Purpose

Lessen network traffic by combining repeated keys in MapTask.

Important Functions

```
void reduce(WritableComparable key, Iterator values,
            OutputCollector output, Reporter reporter);
```

Example Implementation

- `LongSumReducer` adds `LongWritable` values

Behavior

- Framework decides when to call.
- Uses `Reducer` interface, but called with partial list of values.

Extended Combining

Problem

- 1000 map outputs are buffered before combining.
- Keys can still be repeated enough to unbalance a reduce.

Two Phase Reduce

1. Run a MapReduce to combine values
   - Use `Partitioner` to balance a key over Reducers
   - Run Combiner in Mapper and Reducer
2. Run a MapReduce to reduce values
   - Map with `IdentityMapper`
   - Partition normally
   - Reduce normally

General Advice

Small Work Units

- More inputs than Mappers
- Ideally, more reduce tasks than Reducers
- Too many tasks increases overhead
- Aim for constant-memory Mappers and Reducers

Map Only

- Skip `IdentityReducer` by setting `numReduceTasks` to -1

Outside Tables

- Increase HDFS replication before launching
- Keep random access tables in memory
- Use multithreading to share memory

Netflix data

Goal

Find similar movies from ratings provided by users

Vector Model

- Give each movie a vector
- Make one dimension per user
- Put origin at average rating (so poor is negative)
- Normalize all vectors to unit length
    - Often called cosine similarity

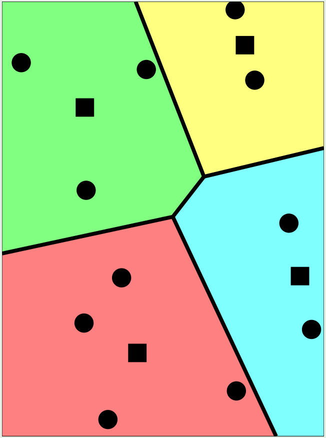
Issues

- Users are biased in the movies they rate

+ Addresses different numbers of raters

k-Means Clustering

Two Dimensional Clusters



d is sum of squares

Goal

Cluster similar data points

Approach

Given data points $x[i]$ and distance d:

- Select k centers c
- Assign $x[i]$ to closest center $c[i]$
- Minimize $\sum_i d(x[i], c[i])$

Lloyd's Algorithm

Algorithm

1. Randomly pick centers, possibly from data points
2. Assign points to closest center
3. Average assigned points to obtain new centers
4. Repeat 2 and 3 until nothing changes

Issues

- Takes superpolynomial time on some inputs
- Not guaranteed to find optimal solution

+ Converges quickly in practice

Lloyd's Algorithm in MapReduce

Reformatting Data

Create a SequenceFile for fast reading. Partition as you see fit.

Initialization

Use a seeded random number generator to pick initial centers.

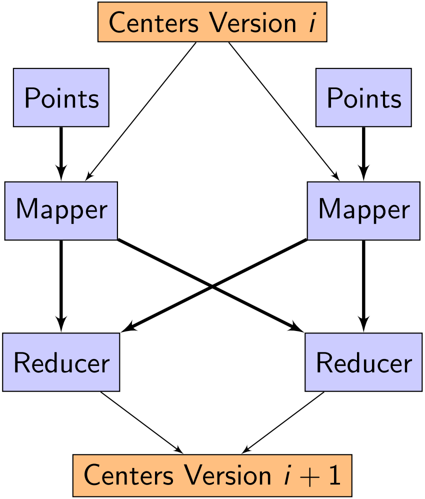
Iteration

Load centers table in MapRunnable or Mapper.

Termination

Use TextOutputFormat to list movies in each cluster.

Iterative MapReduce



Find Nearest Center

Key is Center, Value is Movie

Average Ratings

Direct Implementation

Mapper

- Load all centers into RAM off HDFS
- For each movie, measure distance to each center
- Output key identifying the closest center

Reducer

- Output average ratings of movies

Issues

- Brute force distance and all centers in memory
- Unbalanced reduce, possibly even for large k

Two Phase Reduce

Implementation

1. Combine
   - Mapper key identifies closest center, value is point.
   - Partitioner balances centers over reducers.
   - Combiner and Reducer add and count points.
2. Recenter
   - `IdentityMapper`
   - Reducer averages values

Issues

+ Balanced reduce

- Two phases

- Mapper still has all k centers in memory

Large *k*

Implementation

- Map task responsible for part of movies and part of *k* centers.
  - For each movie, finds closest of known centers.
  - Output key is point, value identifies center and distance.
- Reducer takes minimum distance center.
  - Output key identifies center, value is movie.
- Second phase averages points in each center.

Issues

+ Large *k* while still fitting in RAM
- Reads data points multiple times
- Startup and intermediate storage costs

Exercises

k-Means

- Run on part of Netflix to cluster movies
- Read about and implement Canopies: http://www.kamalnigam.com/papers/canopy-kdd00.pdf