MapReduce Features

Contents

- Counters
- Joins
- Distributed Cache

Counters

- Counters are a useful channel for gathering statistics about the job
  - like number of rows read, number of rows written as output etc.
  - A Counter is generally used to keep track of occurrences of any events.
  - Useful for problem diagnosis
    - # of invalid records
  - Easier to use and to retrieve than logging

- Typically some of the operations of Hadoop counters are:
  - Number of mapper and reducer launched.
  - The number of bytes was read and written
  - The number of tasks was launched and successfully ran
  - The amount of CPU and memory consumed is appropriate or not for your job and cluster nodes

Built in counter groups

- Built-in counters
  - Report various metrics for Map-Reduce jobs

Group	Name/Enum
MapReduce task counters	`org.apache.hadoop.mapreduce.TaskCounter`
Filesystem counters	`org.apache.hadoop.mapreduce.FileSystemCounter`
`FileInputFormat` counters	`org.apache.hadoop.mapreduce.lib.input.FileInputFormatCounter`
`FileOutputFormat` counters	`org.apache.hadoop.mapreduce.lib.output.FileOutputFormatCounter`
Job counters	`org.apache.hadoop.mapreduce.JobCounter`

Built in Map Reduce task counter -1

Counter	Description
Map input records (MAP_INPUT_RECORDS)	The number of input records consumed by all the maps in the job. Incremented every time a record is read from a RecordReader and passed to the map's map() method by the framework.
Map output records (MAP_OUTPUT_RECORDS)	The number of map output records produced by all the maps in the job. Incremented every time the collect() method is called on a map's OutputCollector.
Map output bytes (MAP_OUTPUT_BYTES)	The number of bytes of uncompressed output produced by all the maps in the job. Incremented every time the collect() method is called on a map's OutputCollector.
Combine input records (COMBINE_INPUT_RECORDS)	The number of input records consumed by all the combiners (if any) in the job. Incremented every time a value is read from the combiner's iterator over values. Note that this count is the number of values consumed by the combiner, not the number of distinct key groups
Combine output records (COMBINE_OUTPUT_RECORDS)	The number of output records produced by all the combiners (if any) in the job. Incremented every time the collect() method is called on a combiner's OutputCollector.

Built in Map Reduce task counter-2

Counter	Description
Reduce input records (REDUCE_INPUT_RECORDS)	The number of input records consumed by all the reducers in the job. Incremented every time a value is read from the reducer's iterator over values. If reducers consume all of their inputs, this count should be the same as the count for map output records.
Reduce output records (REDUCE_OUTPUT_RECORDS)	The number of reduce output records produced by all the maps in the job. Incremented every time the `collect()` method is called on a reducer's `OutputCollector`.
Reduce shuffle bytes (REDUCE_SHUFFLE_BYTES)	The number of bytes of map output copied by the shuffle to reducers.
Spilled records (SPILLED_RECORDS)	The number of records spilled to disk in all map and reduce tasks in the job.
CPU milliseconds (CPU_MILLISECONDS)	The cumulative CPU time for a task in milliseconds, as reported by */proc/cpuinfo*.
Physical memory bytes (PHYSICAL_MEMORY_BYTES)	The physical memory being used by a task in bytes, as reported by */proc/meminfo*.

Built in Map Reduce task counter-3

Counter	Description
Virtual memory bytes (VIRTUAL_MEMORY_BYTES)	The virtual memory being used by a task in bytes, as reported by */proc/meminfo*.
Committed heap bytes (COMMITTED_HEAP_BYTES)	The total amount of memory available in the JVM in bytes, as reported by `Runtime.getRuntime().totalMemory()`.
GC time milliseconds (GC_TIME_MILLIS)	The elapsed time for garbage collection in tasks in milliseconds, as reported by `GarbageCollectorMXBean.getCollectionTime()`.
Shuffled maps (SHUFFLED_MAPS)	The number of map output files transferred to reducers by the shuffle
Failed shuffle (FAILED_SHUFFLE)	The number of map output copy failures during the shuffle.
Merged map outputs (MERGED_MAP_OUTPUTS)	The number of map outputs that have been merged on the reduce side of the shuffle.

Built in file system task counter

Counter	Description
Filesystem bytes read (`BYTES_READ`)	The number of bytes read by the filesystem by map and reduce tasks. There is a counter for each filesystem, and *Filesystem* may be Local, HDFS, S3, etc.
Filesystem bytes written (`BYTES_WRITTEN`)	The number of bytes written by the filesystem by map and reduce tasks.
Filesystem read ops (`READ_OPS`)	The number of read operations (e.g., open, file status) by the filesystem by map and reduce tasks.
Filesystem large read ops (`LARGE_READ_OPS`)	The number of large read operations (e.g., list directory for a large directory) by the filesystem by map and reduce tasks.
Filesystem write ops (`WRITE_OPS`)	The number of write operations (e.g., create, append) by the filesystem by map and reduce tasks.

Built in Counter

Built-in FileInputFormat task counters

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

Built-in FileOutputFormat task counters

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

Built in Job counter

Counter	Description
Launched map tasks (TOTAL_LAUNCHED_MAPS)	The number of map tasks that were launched. Includes tasks that were started speculatively
Launched reduce tasks (TOTAL_LAUNCHED_REDUCES)	The number of reduce tasks that were launched. Includes tasks that were started speculatively.
Launched uber tasks (TOTAL_LAUNCHED_UBERTASKS)	The number of uber tasks that were launched.
Maps in uber tasks (NUM_UBER_SUBMAPS)	The number of maps in uber tasks.
Reduces in uber tasks (NUM_UBER_SUBREDUCES)	The number of reduces in uber tasks.
Failed map tasks (NUM_FAILED_MAPS)	The number of map tasks that failed.
Failed reduce tasks (NUM_FAILED_REDUCES)	The number of reduce tasks that failed.
Failed uber tasks (NUM_FAILED_UBERTASKS)	The number of uber tasks that failed.
Killed map tasks (NUM_KILLED_MAPS)	The number of map tasks that were killed.

Built in job counter

Killed reduce tasks (NUM_KILLED_REDUCES)	The number of reduce tasks that were killed.
Data-local map tasks (DATA_LOCAL_MAPS)	The number of map tasks that ran on the same node as their input data.
Rack-local map tasks (RACK_LOCAL_MAPS)	The number of map tasks that ran on a node in the same rack as their input data, but were not data-local.
Other local map tasks (OTHER_LOCAL_MAPS)	The number of map tasks that ran on a node in a different rack to their input data. Inter-rack bandwidth is scarce, and Hadoop tries to place map tasks close to their input data, so this count should be low.
Total time in map tasks (MILLIS_MAPS)	The total time taken running map tasks, in milliseconds. Includes tasks that were started speculatively. See also corresponding counters for measuring core and memory usage (VCORES_MILLIS_MAPS and MB_MILLIS_MAPS).
Total time in reduce tasks (MILLIS_REDUCES)	The total time taken running reduce tasks, in milliseconds. Includes tasks that were started speculatively. See also corresponding counters for measuring core and memory usage (VCORES_MILLIS_REDUCES and MB_MILLIS_REDUCES).

User-Defined Java Counter

- MapReduce allows user code to define a set of counters, which are then incremented as desired in the mapper or reducer
- Counters are defined by a Java enum
  - The name of the enum is the group name
  - The enum's fields are the counter names

```
enum Temperature {
  MISSING,
  MALFORMED
}
```

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

  parser.parse(value);
  if (parser.isValidTemperature()) {
    int airTemperature = parser.getAirTemperature();
    output.collect(new Text(parser.getYear()),
        new IntWritable(airTemperature));
  } else if (parser.isMalformedTemperature()) {
    System.err.println("Ignoring possibly corrupt input: " + value);
    reporter.incrCounter(Temperature.MALFORMED, 1);
  } else if (parser.isMissingTemperature()) {
    reporter.incrCounter(Temperature.MISSING, 1);
  }

  // dynamic counter
  reporter.incrCounter("TemperatureQuality", parser.getQuality(), 1);

}
```

User-Defined Java Counter

```
public class MaxTemperatureWithCounters extends Configured implements Tool {

  enum Temperature {
    MISSING,
    MALFORMED
  }
```

User-Defined Java Counter

```
static class MaxTemperatureMapperWithCounters
    extends Mapper<LongWritable, Text, Text, IntWritable> {

  private NcdcRecordParser parser = new NcdcRecordParser();

  @Override
  protected void map(LongWritable key, Text value, Context context)
      throws IOException, InterruptedException {

    parser.parse(value);
    if (parser.isValidTemperature()) {
      int airTemperature = parser.getAirTemperature();
      context.write(new Text(parser.getYear()),
          new IntWritable(airTemperature));
    } else if (parser.isMalformedTemperature()) {
      System.err.println("Ignoring possibly corrupt input: " + value);
      context.getCounter(Temperature.MALFORMED).increment(1);
    } else if (parser.isMissingTemperature()) {
      context.getCounter(Temperature.MISSING).increment(1);
    }

    // dynamic counter
    context.getCounter("TemperatureQuality", parser.getQuality()).increment(1);
  }
}
```

User-Defined Java Counter

- When the job has successfully completed, it prints out the counters at the end

```
09/04/20 06:33:36 INFO mapred.JobClient:   TemperatureQuality
09/04/20 06:33:36 INFO mapred.JobClient:     2=1246032
09/04/20 06:33:36 INFO mapred.JobClient:     1=973422173
09/04/20 06:33:36 INFO mapred.JobClient:     0=1
09/04/20 06:33:36 INFO mapred.JobClient:     6=40066
09/04/20 06:33:36 INFO mapred.JobClient:     5=158291879
09/04/20 06:33:36 INFO mapred.JobClient:     4=10764500
09/04/20 06:33:36 INFO mapred.JobClient:     9=66136858
09/04/20 06:33:36 INFO mapred.JobClient:   Air Temperature Records
09/04/20 06:33:36 INFO mapred.JobClient:     Malformed=3
09/04/20 06:33:36 INFO mapred.JobClient:     Missing=66136856
```

- Readable names of counters
  - Create a properties file named after the enum, using an underscore as a separator for nested classes

MaxTemperatureWithCounters_Temperature.properties

```
CounterGroupName=Air Temperature Records
MISSING.name=Missing
MALFORMED.name=Malformed
```

Retrieving counter

```
Counters counters = job.getCounters();
long missing = counters.findCounter(
    MaxTemperatureWithCounters.Temperature.MISSING).getValue();
long total = counters.findCounter(TaskCounter.MAP_INPUT_RECORDS).getValue();
```

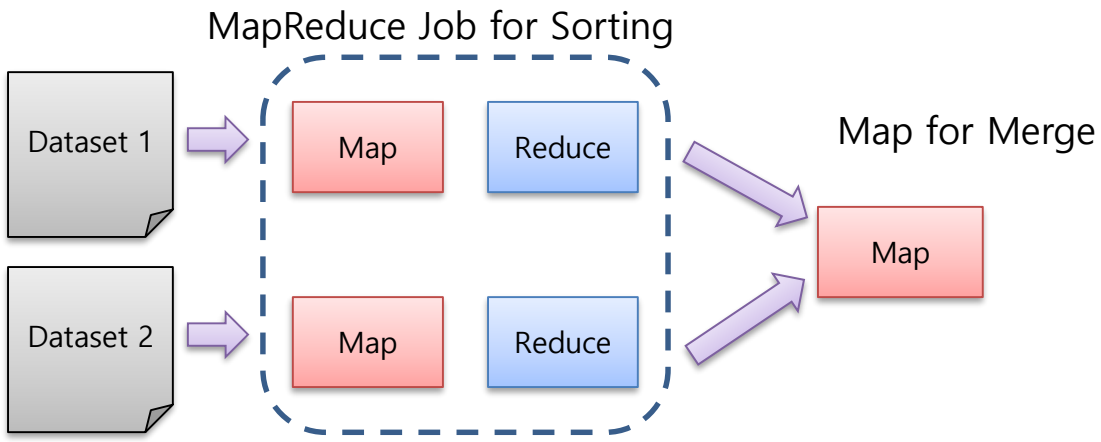
Map-Side Join

- When the join is performed by the mapper, it is called as map-side join.
- In this, the join is performed before data is actually consumed by the map f unction.
- It is mandatory that the input to each map is in the form of a partition and is in sorted order.
- Also, there must be an equal number of partitions and it must be sorted by the join key.
- Requirements
  - Each input dataset must be divided into the same number of partitions
  - It must be sorted by the same key (the join key) in each source
  - All the records for a particular key must reside in the same partition

- Above requirements actually fit the description of the output of a MapReduce job
  - A map-side join can be used to join the outputs of several jobs that had the same number of reducers, the same keys, and output files that are not splittable

Map-Side Join

- Use a CompositeInputFormat from the org.apache.hadoop.mapred.join package to run a map-side join

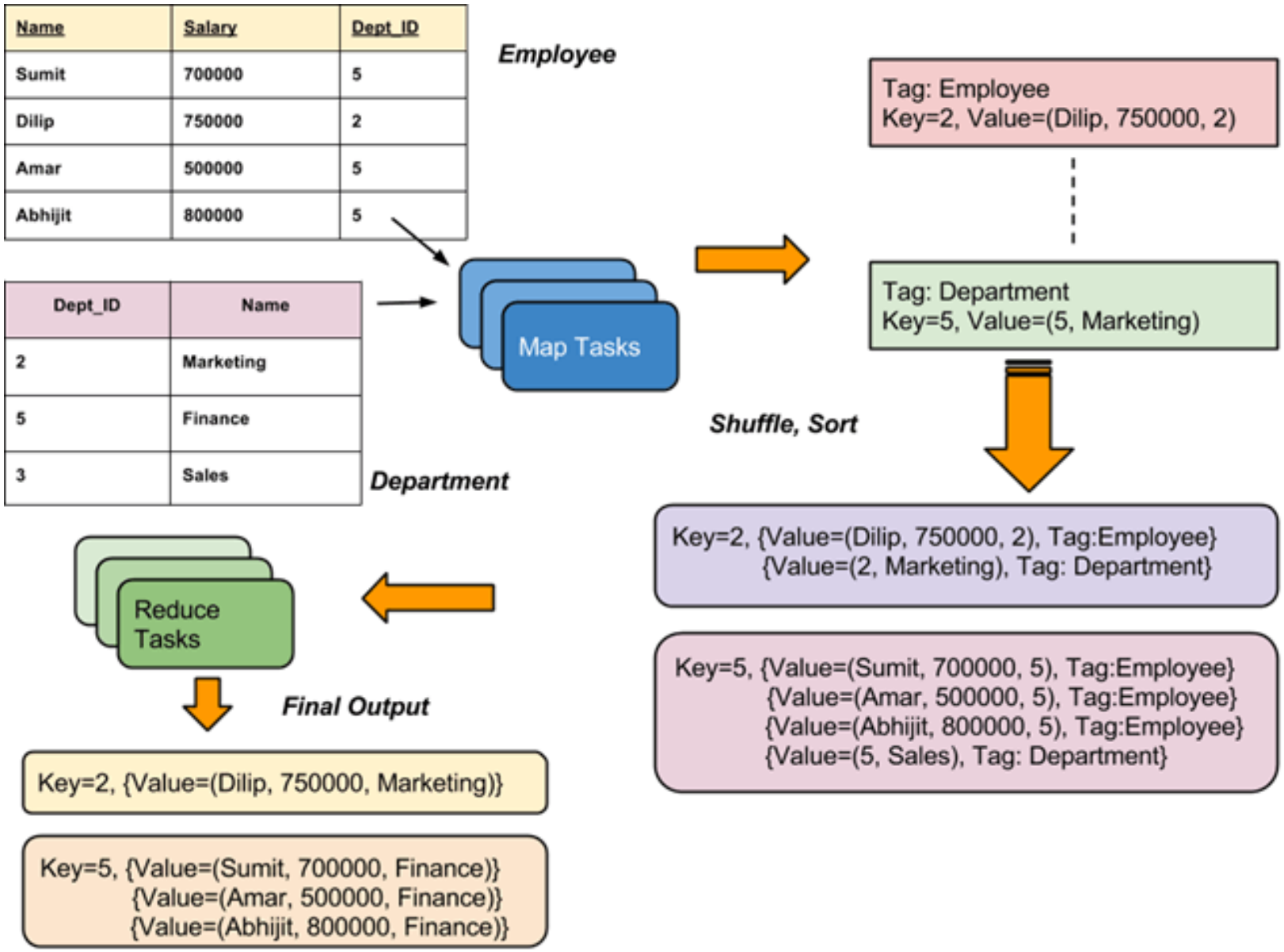


Reduce-Side Join

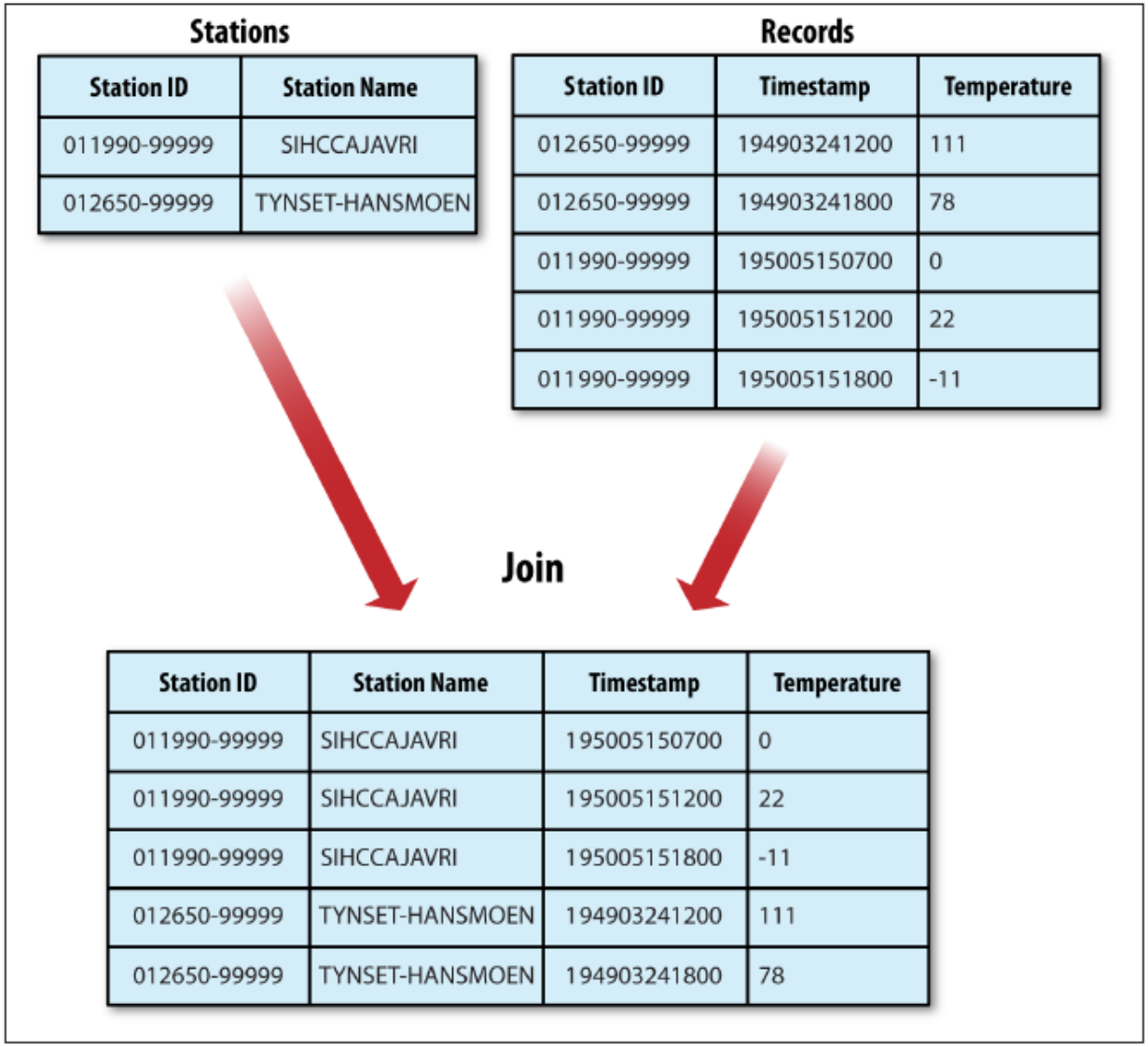
- When the join is performed by the reducer, it is called as reduce-side join.
- There is no necessity in this join to have a dataset in a structured form (or partitioned).
- Here, map side processing emits join key and corresponding tuples of both the tables.
- As an effect of this processing, all the tuples with same join key fall into the same reducer which then joins the records with same join key.
- More general than a map-side join
  - Input datasets don't have to be structured in any particular way
  - Less efficient as both datasets have to go through the MapReduce shuffle

- Idea
  - The mapper tags each record with its source
  - Uses the join key as the map output key so that the records with the same key are brought together in the reducer

Map | Reduce-Side Join



Map | Reduce-Side Join



Stations

Station ID	Station Name
011990-99999	SIHCCAJAVRI
012650-99999	TYNSET-HANSMOEN

Records

Station ID	Timestamp	Temperature
012650-99999	194903241200	111
012650-99999	194903241800	78
011990-99999	195005150700	0
011990-99999	195005151200	22
011990-99999	195005151800	-11

Join

Station ID	Station Name	Timestamp	Temperature
011990-99999	SIHCCAJAVRI	195005150700	0
011990-99999	SIHCCAJAVRI	195005151200	22
011990-99999	SIHCCAJAVRI	195005151800	-11
012650-99999	TYNSET-HANSMOEN	194903241200	111
012650-99999	TYNSET-HANSMOEN	194903241800	78

The Distributed Cache: Motivation

- **A common requirement is for a Mapper or Reducer to need access to some 'side data'**
  - Lookup tables
  - Dictionaries
  - Standard configuration values
- **One option: read directly from HDFS in the `setup` method**
  - Using the API seen in the previous section
  - Works, but is not scalable
- **The Distributed Cache provides an API to push data to all slave nodes**
  - Transfer happens behind the scenes before any task is executed
  - Data is only transferred once to each node, rather
  - Note: Distributed Cache is read-only
  - Files in the Distributed Cache are automatically deleted from slave nodes when the job finishes

Using the Distributed Cache: The Difficult Way

- **Place the files into HDFS**
- **Configure the Distributed Cache in your driver code**

```
Configuration conf = new Configuration();
DistributedCache.addCacheFile(new URI("/myapp/lookup.dat"),conf);
DistributedCache.addFileToClassPath(new Path("/myapp/mylib.jar"),conf);
DistributedCache.addCacheArchive(new URI("/myapp/map.zip",conf));
DistributedCache.addCacheArchive(new URI("/myapp/mytar.tar",conf));
DistributedCache.addCacheArchive(new URI("/myapp/mytgz.tgz",conf));
DistributedCache.addCacheArchive(new URI("/myapp/mytargz.tar.gz",conf));
```

  - `.jar` files added with `addFileToClassPath` will be added to your Mapper or Reducer's classpath
  - Files added with `addCacheArchive` will automatically be dearchived/decompressed

Using the DistributedCache: The Easy Way

- **If you are using ToolRunner, you can add files to the Distributed Cache directly from the command line when you run the job**
  - No need to copy the files to HDFS first
- **Use the `-files` option to add files**

```
hadoop jar myjar.jar MyDriver -files file1, file2, file3, ...
```

- **The `-archives` flag adds archived files, and automatically unarchives them on the destination machines**
- **The `-libjars` flag adds jar files to the classpath**

Accessing Files in the Distributed Cache

- **Files added to the Distributed Cache are made available in your task's local working directory**
  - Access them from your Mapper or Reducer the way you would read any ordinary local file

```
File f = new File("file_name_here");
```

Side Data Distribution

- Side data
  - Extra read-only data needed by a job to process the main dataset
- The challenge is to make side data available to all the map or reduce tasks (which are spread across the cluster)
  - Cache in memory in a static field
  - Using the Job Configuration
  - Distributed Cache

Using the Job Configuration

- Set arbitrary key-value pairs in the job configuration using the various setter methods on **JobConf**
- Useful if one needs to pass a small piece of metadata to tasks
- Don't use this mechanism for transferring more than a few kilobytes of data
  - The job configuration is read by the jobtracker, the tasktracker, and the child JVM, and each time the configuration is read, all of its entries are read into memory, even if they are not used

Distributed Cache

- Distribute datasets using Hadoop's distributed cache mechanism
  - Provides a service for copying files and archives to the task nodes in time for the tasks to use them when they run

- GenericOptionsParser
  - Specify the files to be distributed as a comma-separated list of URIs as the argument to the -files option

```
% hadoop jar job.jar MaxTemperatureByStationNameUsingDistributedCacheFile \
  -files input/ncdc/metadata/stations-fixed-width.txt input/ncdc/all output
```

  - This command will copy the local file *stations-fixed-width.txt* to the task nodes

Distributed Cache

- Hadoop copies the file specified by the –file and -archives options to the jobtracker's filesystem (normally HDFS)
- Before a task is run, the tasktracker copies the files from the jobtracker's filesystem to a local disk
- The tasktracker also maintains a reference count for the number of tasks using each file in the cache
- After the task has run, the file's reference count is decreased by one, and when it reaches zero it is eligible for deletion
- Files are deleted to make room for a new file when the cache exceeds a certain size—10 GB by default