MapReduce Features

Counters

Built-in Counters

Hadoop comes with some built-in counters to report various metrics for your job.

Table 8-1. Built-in counter groups

Group	Name/Enum
MapReduce Task Counters	`org.apache.hadoop.mapred.Task$Counter` (0.20) `org.apache.hadoop.mapreduce.TaskCounter` (post 0.20)
Filesystem Counters	`FileSystemCounters` (0.20) `org.apache.hadoop.mapreduce.FileSystemCounter` (post 0.20)
FileInputFormat Counters	`org.apache.hadoop.mapred.FileInputFormat$Counter` (0.20) `org.apache.hadoop.mapreduce.lib.input.FileInputFormatCounter` (post 0.20)
FileOutputFormat Counters	`org.apache.hadoop.mapred.FileOutputFormat$Counter` (0.20) `org.apache.hadoop.mapreduce.lib.output.FileOutputFormatCounter` (post 0.20)
Job Counters	`org.apache.hadoop.mapred.JobInProgress$Counter` (0.20) `org.apache.hadoop.mapreduce.JobCounter` (post 0.20)

Task Counters

Gather information about tasks over their execution and are maintained by each task attempt. Periodically they are sent to the jobtracker or, if YARN is used, they are fully sent.

Table 8-2. Built-in MapReduce task counters

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 skipped records (MAP_SKIPPED_RECORDS)	The number of input records skipped by all the maps in the job. See "Skipping Bad Records" on page 217.
Map input bytes (MAP_INPUT_BYTES)	The number of bytes of uncompressed input 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.

Split raw bytes (SPLIT_RAW_BYTES)	The number of bytes of input split objects read by maps. These objects represent the split metadata (that is, the offset and length within a file) rather than the split data itself, so the total size should be small.
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`.
Map output materialized bytes (MAP_OUTPUT_MATERIALIZED_BYTES)	The number of bytes of map output actually written to disk. If map output compression is enabled this is reflected in the counter value.
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 (which would not be a useful metric, since there is not necessarily one group per key for a combiner; see "Combiner Functions" on page 34, and also "Shuffle and Sort" on page 205).
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`.
Reduce input groups (REDUCE_INPUT_GROUPS)	The number of distinct key groups consumed by all the reducers in the job. Incremented every time the reducer's `reduce()` method is called by the framework.
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 skipped groups (REDUCE_SKIPPED_GROUPS)	The number of distinct key groups skipped by all the reducers in the job. See "Skipping Bad Records" on page 217.
Reduce skipped records (REDUCE_SKIPPED_RECORDS)	The number of input records skipped by all the reducers in the job.
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*.
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()`. From 0.21.
Shuffled maps (SHUFFLED_MAPS)	The number of map output files transferred to reducers by the shuffle (see "Shuffle and Sort" on page 205). From 0.21.
Failed shuffle (FAILED_SHUFFLE)	The number of map output copy failures during the shuffle. From 0.21.

(FAILED_SHUFFLE)	
Merged map outputs (MERGED_MAP_OUTPUTS)	The number of map outputs that have been merged on the reduce side of the shuffle. From 0.21.

Table 8-3. Built-in filesystem task counters

Counter	Description
Filesystem bytes read (BYTES_READ)	The number of bytes read by each filesystem by map and reduce tasks. There is a counter for each filesystem: *Filesystem* may be Local, HDFS, S3, KFS, etc.
Filesystem bytes written (BYTES_WRITTEN)	The number of bytes written by each filesystem by map and reduce tasks.

Table 8-4. Built-in FileInputFormat task counters

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

Table 8-5. 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.

Job Counters

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 (see "YARN (MapReduce 2)" on page 194) that were launched. From 0.23.
Maps in uber tasks (NUM_UBER_SUBMAPS)	The number of maps in uber tasks. From 0.23.
Reduces in uber tasks (NUM_UBER_SUBREDUCES)	The number of reduces in uber tasks. From 0.23.
Failed map tasks (NUM_FAILED_MAPS)	The number of map tasks that failed. See "Task Failure" on page 200 for potential causes.
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. From 0.23.
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 that are 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. See Figure 2-2.
Total time in map tasks (SLOTS_MILLIS_MAPS)	The total time taken running map tasks in milliseconds. Includes tasks that were started speculatively.
Total time in reduce tasks (SLOTS_MILLIS_REDUCES)	The total time taken running reduce tasks in milliseconds. Includes tasks that were started speculatively.
Total time in map tasks waiting after reserving slots (FALLOW_SLOTS_MILLIS_MAPS)	The total time spent waiting after reserving slots for map tasks in milliseconds. Slot reservation is Capacity Scheduler feature for high-memory jobs, see "Task memory limits" on page 316. Not used by YARN-based MapReduce.
Total time in reduce tasks waiting after reserving slots (FALLOW_SLOTS_MILLIS_REDUCES)	The total time spent waiting after reserving slots for reduce tasks in milliseconds. Slot reservation is Capacity Scheduler feature for high-memory jobs, see "Task memory limits" on page 316. Not used by YARN-based MapReduce.

User Defined Java Counters

Defined by a **Enum** and using a **Reporter.incrCounter()** object that can increment the counters.

Readable counter names

By default, a counter's name is the enum's fully qualified Java classname. To change it, you must create a properties file that must contain a single property named **CounterGroupName** with the value to display.

The file must be named using an underscore as separator for nested classes and it must be placed in the same directory as the top-level class containing the *enum*.

Retrieving counters

Using **hadoop job -counter** you can retrieve counter values.

User-Defined Streaming Counters

A Streaming program can also increment a counter by sending a formatted line to the standard error stream with the following format:

```
reporter:counter:group,counter,amount
```

In Python:

```
    sys.stderr.write("reporter:counter:Temperature,Missing,1\n")

## Sorting
To sort, we **must** sort a numeric key, but with **Text** it doesn't work

### Total sort: Producing a globally sorted file
Using a single partition is inneficient. The way is to produce a set of sorted files that concaten

| Temperature           | -10   | -10-0 | 0-10 | +10 |
| Proportion of records | 11%   | 13%   | 17%  | 59% |

This are not even. *Sampling* the key space to look at a small subset of the keys to approximate t

| Temperature           | -5.6  | -5.6-13.9 | 13.9-22 | +22 |
| Proportion of records | 29%   | 24%   | 23%  | 24% |

### Secondary sort
For any particular reducer key, the values are *not* sorted. To sort the values we must change the

For example the year and the temperature, that we want to sort by year and then by temperature. Ne

* Make the key a composite of key and value
* The sort comparator should order by the composite key, that is, the key and value
* The partitioner and grouping comparator for the composite key should consider only the key for p

## Joins
###Map-Side Joins
Performs the join before the data reaches the map function. Each input dataset must be divided int

![Inner join of two datasets](img/image-41.png)
```

```
Use a **CompositeInputFormat**to run a map-side join.

### Reduce-Side Joins
Is less efficient that the map-side. The mapper tags each record with its source and uses the join

* *Multiple inputs*: **MultipleInputs** must be used as the input sources are usually different
* *Secondary sort*: To perform a join it's important to have the data ordered.

## Side Data Distribution
Considered extra read-only data needed by a job to process the main dataset.

### Using the Job Configuration
Arbitrary key/value pairs can be set in the job configuration using the setters for **Configuratio

### Distributed Cache
#### Usage
For tools thatuse the **GenericOptionsParser* you can specify the files to be distributed as a com

Files will be localized under the **${mapred.local.dir}/taskTracker/archive within each TaskTracke

#### The distributed cache API
Putting data with **Job.addCacheXXXX()** and getting with **JobContext.setCacheXXXX()**.

![Cache API](img/image-42.png)

## MapReduce Library Classes
Hadoop comes with a library of mappers and reducers for commonly used functions:

![MapReduce Library classes](img/image-43.png)
```