How MapReduce works?

Cloudwick Technologies

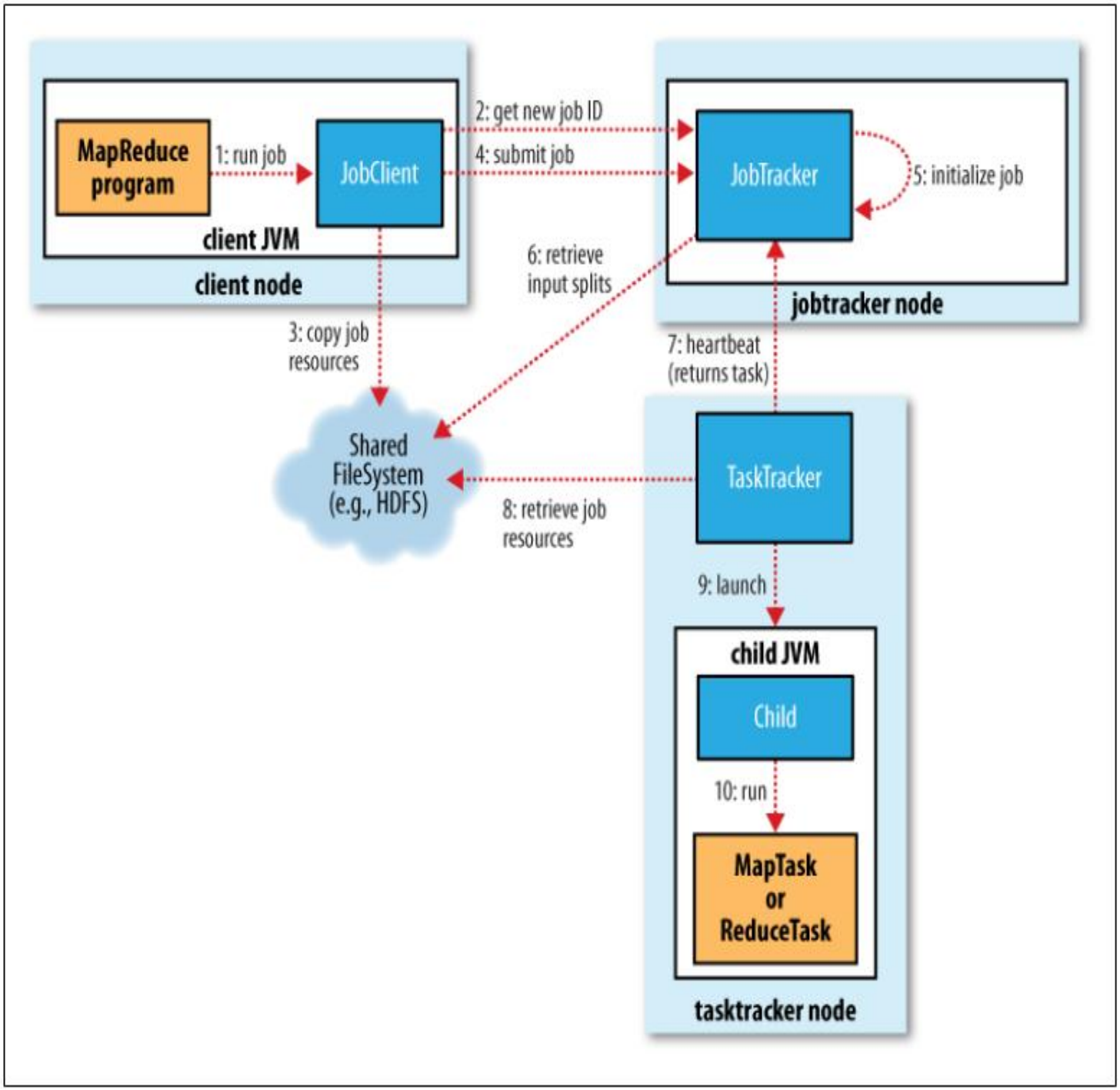
Agenda

- Anatomy of MapReduce Job Run
- Job Submission
- Job Initialization
- Task Assignment
- Job Completion
- Job Scheduling
- Job Failures
- Shuffle and Sort
- Oozie Workflow

Cloudwick Technologies

Anatomy of MapReduce (Classic)

- The client submits the MapReduce job.
- The jobtracker, which coordinates the job run. The jobtracker is a Java application whose main class is JobTracker.
- The tasktrackers, which run the tasks that the job has been split into. Tasktrackers are Java applications whose main class is TaskTracker.
- The distributed filesystem, which is used for sharing job files between the other entities.



Cloudwick Technologies

Job Submission

The submit() method on Job creates an internal JobSummitter instance and calls sub mitJobInternal() on it. Having submitted the job, waitForCompletion() polls the job's progress once a second and reports the progress to the console if it has changed since the last report.

- Asks the jobtracker for a new job ID (by calling getNewJobId() on JobTracker) (step 2).
- Checks the output specification of the job. For example, if the output directory has not been specified or it already exists, the job is not submitted and an error is thrown to the MapReduce program.
- Computes the input splits for the job. If the splits cannot be computed, because the input paths don't exist, for example, then the job is not submitted and an error is thrown to the MapReduce program.
- Copies the resources needed to run the job, including the job JAR file, the configuration file, and the computed input splits, to the jobtracker's file system in a directory named after the job ID. (Step3)
- Tells the jobtracker that the job is ready for execution (by calling submitJob() on JobTracker) (step 4)

Job Initialization

- When the JobTracker receives a call to its submitJob() method, it puts it into an internal queue from where the job scheduler will pick it up and initialize it.
- Initialization involves creating an object to represent the job being run, which encapsulates its tasks, and bookkeeping information to keep track of the tasks' status and progress (step 5).
- To create the list of tasks to run, the job scheduler first retrieves the input splits computed by the client from the shared filesystem (step 6).
- It then creates one map task for each split.
- In addition to the map and reduce tasks, two further tasks are created: a job setup task and a job cleanup task. These are run by tasktrackers and are used to run code to setup the job before any map tasks run, and to cleanup after all the reduce tasks are complete.
- For the job setup task it will create the final output directory for the job and the temporary working space for the task output, and for the job cleanup task it will delete the temporary working space for the task output.

Task Assignment

- Tasktrackers run a simple loop that periodically sends heartbeat method calls to the jobtracker. Heartbeats tell the jobtracker that a tasktracker is alive.

- As a part of the heartbeat, a tasktracker will indicate whether it is ready to run a new task, and if it is, the jobtracker will allocate it a task, which it communicates to the tasktracker using the heartbeat return value (step 7).

- Jobs and tasks are executed as per the Job Scheduling mechanism chosen.

- Map tasks are scheduled as data local and Reduce tasks does not need to be data local.

- Some tasks may or may not be be data local /rack local

- Job Counters info will provide this info.

Task Execution

- First, Task Tracker localizes the job JAR by copying it from the shared filesystem to the tasktracker's
- filesystem.

- It also copies any files needed from the distributed cache by the application to the local disk (step 8)

- Second, it creates a local working directory for the task, and un-jars the contents of the JAR into this
- directory.

- Third, it creates an instance of TaskRunner to run the task.

- TaskRunner launches a new Java Virtual Machine (step 9) to run each task in (step 10), so that any bugs in the user-defined map and reduce functions don't affect the tasktracker.

- Child Process communicates with its parent for progress

- Each task can perform setup and cleanup actions, which are run in the same JVM as the task itself, and are determined by the OutputCommitter for the job

- The cleanup action is used to commit the task, which in the case of file-based jobs means that its output is written to the final location for that task.

Cloudwick Technologies

Progress of MapReduce

All of the following operations constitute progress:

- Reading an input record (in a mapper or reducer)
- Writing an output record (in a mapper or reducer)
- Setting the status description on a reporter (using Reporter's setStatus() method)
- Incrementing a counter (using Reporter's incrCounter() method)
- Calling Reporter's progress() method

Cloudwick Technologies

Job Completion

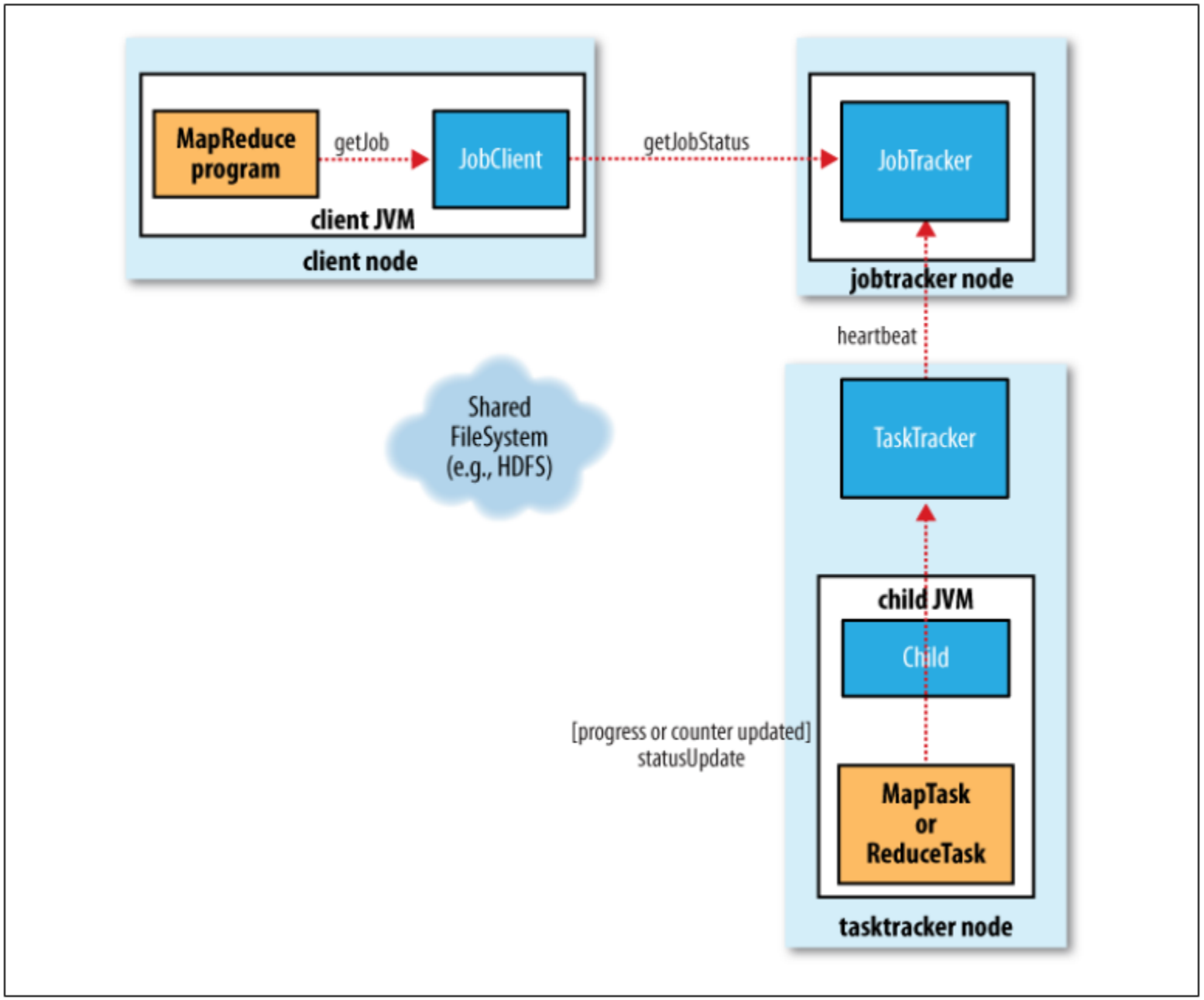
- When the jobtracker receives a notification that the last task for a job is complete (this will be the special job cleanup task), it changes the status for the job to “successful.” Then, when the Job polls for status, it learns that the job has completed successfully, so it prints a message to tell the user and then returns from the waitForCompletion() method.

- Last, the jobtracker cleans up its working state for the job and instructs tasktrackers to do the same (so intermediate output is deleted, for example).

Cloudwick Technologies

YARN (MapReduce 2)

For very large clusters in the region of 4000 nodes and higher, the MapReduce system described in classic MapReduce begins to hit scalability bottlenecks. YARN can be used to overcome.



Cloudwick Technologies

MapReduce on YARN involves more entities than classic MapReduce. They are:

- The client, which submits the MapReduce job.
- The YARN resource manager, which coordinates the allocation of compute resources on the cluster.
- The YARN node managers, which launch and monitor the compute containers on machines in the cluster.
- The MapReduce application master, which coordinates the tasks running the MapReduce job. The application master and the MapReduce tasks run in containers that are scheduled by the resource manager, and managed by the node managers.
- The distributed filesystem, which is used for sharing job files between the other entities.

Job Scheduling

FIFO Scheduler: Default scheduler which will process all tasks of a job before start executing next job tasks.

Fair Scheduler : Allows multiple users of cluster a fair share simultaneously. Each job is assigned with a pool and each pool is assigned with an even share of available task slots.

Capacity Scheduler: Support multiple queues. Queues are guaranteed a fraction of the capacity of the grid (their 'guaranteed capacity') in the sense that a certain capacity of resources will be at their disposal. All jobs submitted to a queue will have access to the capacity guaranteed to the queue.

Job Failures

3 Types of failures (Task, TaskTracker and JobTracker)

Task Failure:

- Usually this happens when user code in the map or reduce task throws a runtime exception. If this happens, the child JVM reports the error back to its parent tasktracker, before it exits. The error ultimately makes it into the user logs. The tasktracker marks the task attempt as *failed, freeing up a slot to run another task.*

- When the jobtracker is notified of a task attempt that has failed (by the tasktracker's heartbeat call), it will reschedule execution of the task. The jobtracker will try to avoid rescheduling the task on a tasktracker where it has previously failed. Furthermore, if a task fails four times (or more), it will not be retried further and job will fail.

- A task attempt may also be *killed, which is different from it failing. A task attempt may* be killed because it is a speculative duplicate ("Speculative Execution"), or because the tasktracker it was running on failed, and the jobtracker marked all the task attempts running on it as killed. Killed task attempts do not count against the number of attempts to run the task (as set by mapred.map.max.attempts and mapred.reduce.max.attempts), since it wasn't the task's fault that an attempt was killed.

Job Failures

Tastracker Failure:

The jobtracker will notice a tasktracker that has stopped sending heartbeats (if it hasn't received one for 10 minutes, configured via the mapred.task tracker.expiry.interval property, in milliseconds) and remove it from its pool of tasktrackers to schedule tasks on. The jobtracker arranges for map tasks that were run and completed successfully on that tasktracker to be rerun if they belong to incomplete jobs, since their intermediate output residing on the failed tasktracker's local filesystem may not be accessible to the reduce task. Any tasks in progress are also rescheduled.

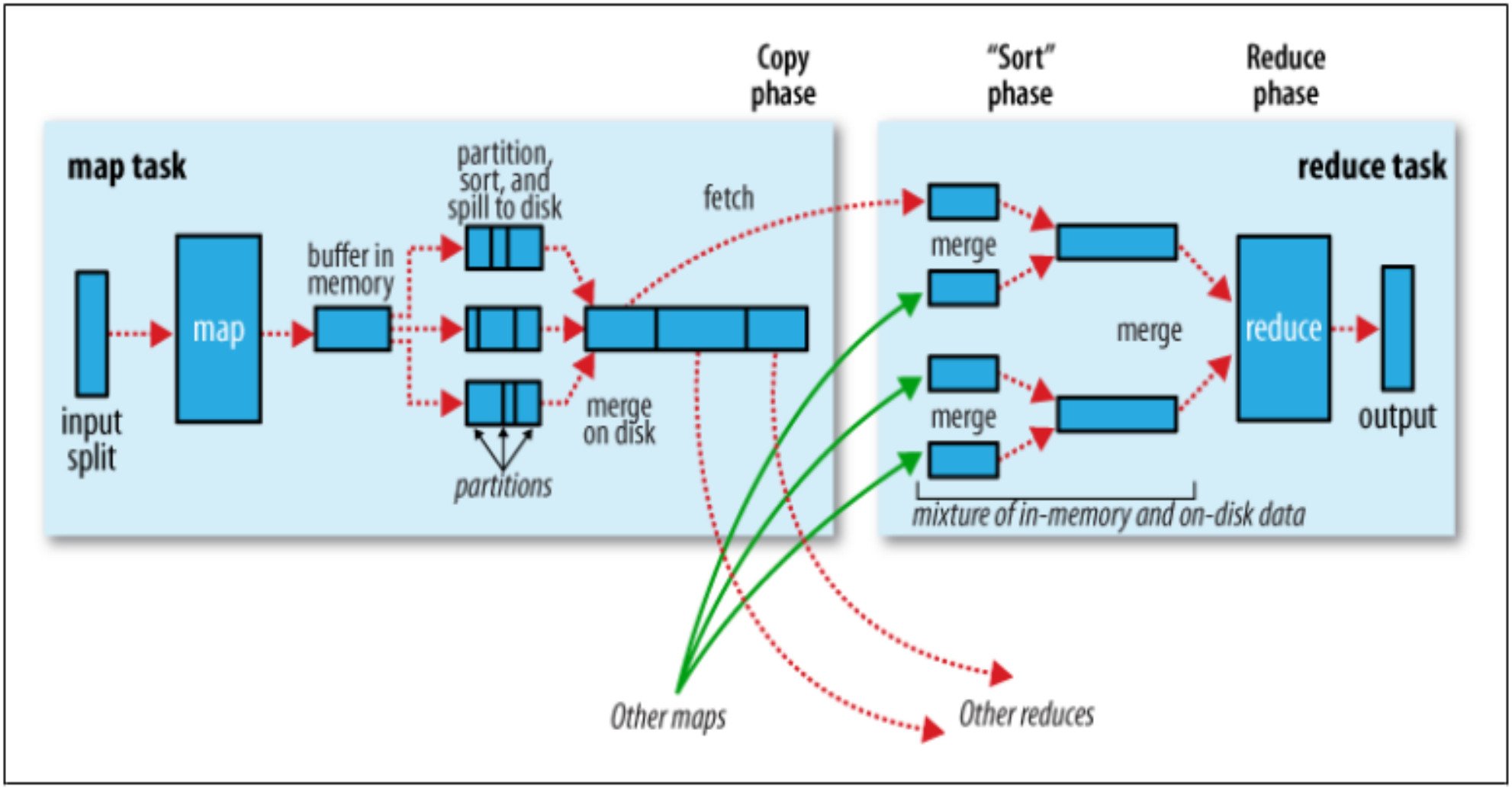
Jobtracker Failure:

Failure of the jobtracker is the most serious failure mode. Hadoop has no mechanism for dealing with failure of the jobtracker—it is a single point of failure—so in this case the job fails. However, this failure mode has a low chance of occurring.

After restarting a jobtracker, any jobs that were running at the time it was stopped will need to be re-submitted. There is a configuration option that attempts to recover any running jobs (mapred.jobtracker.restart.recover, turned off by default), however it is known not to work reliably, so should not be used.

Shuffle and Sort

MapReduce makes the guarantee that the input to every reducer is sorted by key. The process by which the system performs the sort—and transfers the map outputs to the reducers as inputs—is known as the *shuffle*



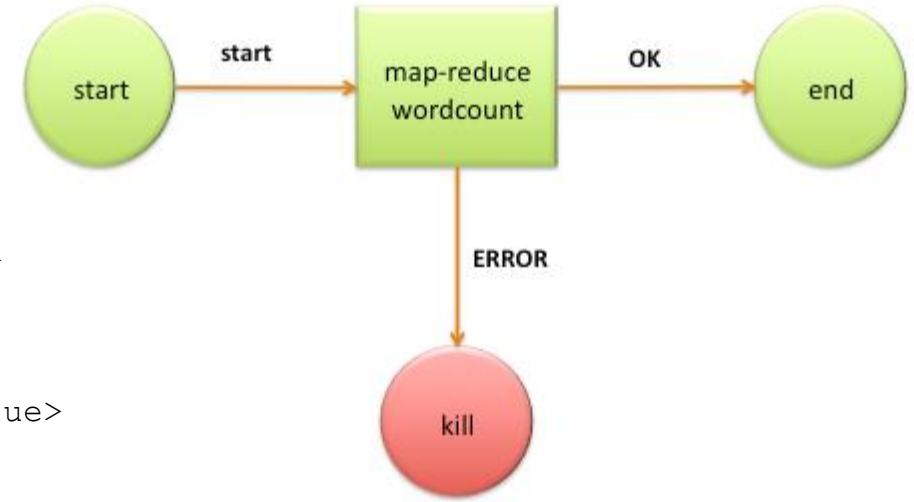
Cloudwick Technologies

Oozie Workflows

- Most of the problems cannot be solved with a single MapReduce job
- So, a workflow of jobs is needed to finish the job.
- If you are running many jobs (MR, Pig, Hive, Sqoop etc) in parallel and want to use the output as input to next job, Oozie workflow is used.
- Jobs can be run at specific times or depending on availability of data or completion of another job.
- Oozie workflows are written in XML.
- Workflow consists of control flow nodes and action nodes
- Control flow nodes define the beginning and end of a workflow
- Action nodes trigger the execution of a processing task

Oozie Workflows

```
<workflow-app name='wordcount-wf' xmlns="uri:oozie:workflow:0.1">
    <start to='wordcount'/>
    <action name='wordcount'>
        <map-reduce>
            <job-tracker>${jobTracker}</job-tracker>
            <name-node>${nameNode}</name-node>
            <configuration>
                <property>
                    <name>mapred.mapper.class</name>
                    <value>org.myorg.WordCount.Map</value>
                </property>
                <property>
                    <name>mapred.reducer.class</name>
                    <value>org.myorg.WordCount.Reduce</value>
                </property>
                <property>
                    <name>mapred.input.dir</name>
                    <value>${inputDir}</value>
                </property>
                <property>
                    <name>mapred.output.dir</name>
                    <value>${outputDir}</value>
                </property>
            </configuration>
        </map-reduce>
        <ok to='end'/>
        <error to='end'/>
    </action>
    <kill name='kill'>
        <message>Something went wrong: ${wf:errorCode('wordcount')}</message>
    </kill/>
    <end name='end'/>
</workflow-app>
```



Cloudwick Technologies

Questions?