How MapReduce works

Date: Page: 42

- Submit() - Run MR Job on Job object.
- waitForCompletion() - Submit Job then wait for finish.
- Conceal great deal of processing

Figure 7.1 (How MR run on Hadoop)

- 5 independent entities

1) Client
2) YARN RM -
3) NM
4) AM
5) Distributed file system

Job Submission

- Submit() on Job create JobSubmitter instance - Calls submitJobInternal() (Step 1)
- waitForCompletion polls Job progress once per second, report progress to Console (if changed). Job counters are displayed when Job complete successfully otherwise error logged to console.
- Ask RM for new App ID. (2)
- Copy resources to HDFS directory named JobID. (JAR, Config file, input splits) (3)
- Submit Job by calling SubmitApplication() on RM (4)

Job Initialization

- When RM receives a Call SubmitApplication() - hands off req to YARN Scheduler
- Scheduler allocate Container (5a)
- RM then launch AM process under NM mgt (5b)

- AM for MR Job is a Java App (main class - MRAppMaster)
- It initializes Job by creating book keeping objects to keep track of Job's progress, receive progress, completion report from task (6)
- Retrieve input splits from HDFS. (7), Create map tasks object, no of reduce task objects, Tasks are given ID.
- AM decide how to run tasks - if Job is Small - AM choose to run tasks in same JVM (Uber task)
  Small Job - less than 10 mapper only one reducer
- AM Call setupJob() - create final O/P dir for Job and temporary working space for task O/P.

Task Assignment

- If not Uber task, AM request containers for all map and Reduce task in Job from RM (8)
- Req for map task made first (Priority) than Reduce task (not made until 5% of map task completed)
- Reduce task can run anywhere but req for map task have data locality constraints
- optimal - task is data local | rack local | diff racks
- Req for memory and CPU for task - M/R task 1024 mb and 1 virtual core (configurable)
- mapreduce.map.memory.mb | CPU.vcores
  ——— .reduce. — .mb | CPU.vcores

Task Execution

- Once task has been assigned resources for a container, AM starts container by contacting NM (9a, 9b)
- Task is executed by Java App (main class YarnChild)
- Before it run task, it localize resources that that task needs (Job Config, JAR file) (10)
- finally, it runs the map or Reduce task (11)
- YARN child ~~is~~ runs in a dedicated JVM, So any bugs (in map Reduce) don't affect the NM.

Streaming

- It runs Special map Reduce tasks (Fig 7.2)
- It Comm with process using standard I/p Streams.
- Java process passes input key/value pairs to external process.
- Run user defined map/Red func and passes output key-value pair back to Java Process.
- NM view — as if child process ran the M/R code itself

Progress and Status updates

- MR Jobs are long running Jobs (los to hours), So imp for user to get feedback how Job is progressing.
- Job and its tasks have status (includes)
  - State of Job/task (running, completed, fail)
  - Progress of M/R
  - Value of Job Counter
  - Status msg or description

When task is running - it keep track of progress

- map task - proportion of Input that has been processed

- Reduce task - complex
- Dividing total progress into 3 parts (phases of shuffle)
- if task run reducer 1/2 of its input task progress
  is 5/6 = 1/3 + 1/3 + 1/3 × 1/2

What Constitutes Progress in MR

- Reading an input record
- writing an output record
- Setting the status description
- Incrementing a Counter
- Calling Reporter's progress() method

- child process comm with parent AM through umbilical interface
- Task reports its progress and status back to its AM
- Client can receive latest status by polling the AM every sec.

Fig 7.3

Job Completion

- When Am receive notification that last task for a job is Complete - it change status - "successful".
- It then print a msg to tell user; return from waitforCompletion()
- Job statistics and counters are printed to console.
- finally on Job Completion AM and taskContainers clean up working state | Job info is archived by JobHistory server

Failure

- Real word user code is buggy, process crash, m/c fail.
- Ragits of Hadoop - ability to handle failure and allow job to complete successfully.
- failure of following entities.

Task failure

(1) - user code (M/R) task throw a runtime Exception.
- task JVM reports errors back to its parent app Master by exits.
- make it in user logs.
- AM marks task as failed, free up the Container so res available to another task.

(2) Another failure mode - Sudden exit of task JVM - JVM bugs cause JVM to exit
- NM notices and inform AM so it can mark as failed

(3) Hanging task - AM hasn't received Progress update - mark task as failed.
- timeout period (10 min) configurable (mapreduce.task.timeout in milliseconds)
- Setting 0 - long running tasks never marked as fail never free up its container - slow down cluster
Task - 4 time retry (mapreduce.map.maxattempts / reduce)
- Task attempt may be killed - speculative Execution

AM failure

- App in YARN retried in event failure.
- Max no of attempt to run MR AM is controlled by mapreduce.am.max-attempts property (def 2)
- YARN impose limit for max no of attempt of YARN app M (AM) after

(yarn.resourcemanager.am.max-attempts)

→ Recovery - AM send Heartbeat to RM, will detect failure
Start new instance of master in new container
RM uses job history to recover state of the tasks

(2) - MR client poll AM for progress report
- if AM fail client need new loc.
- During job initialization client ask RM for AM address and caches so not to overload RM
- if AM fail client experience timeout - go to RM again for new add.

NM failure

- if NM fail / slow stop sending HB to RM (10min)
- so RM remove from pool of nodes to schedule containers
- AM / task on failed NM - recovered using prev sections
- AM arrange map tasks that were completed successfully on failed NM to rerun if they belong to incomplete job. (intermediate o/p of failed NM)
- NM - blacklisted - if not failure for app is high, even if NM has not failed, AM reschedule task on diff node (if more than 3 task fail on NM)

RM failure

- failure - no jobs, task container can be launched.
- SPOF
- HA - active - standby conf. | info about running app is stored in highly Ava state store is (ZK / HDFS)
Standby can recover state of failed RM

- Transition of RM from Standby to Active handled by failover Controller. (def automatic) - Use ZK leader election (only 1 Active RM)

Shuffle and Sort

- MR guarantee input to every reducer is sorted by key
- Shuffle - process - System perform sort - transfer map o/p to reducer as I/p Fig 7.4

Map Side

- Buffering writes in memory
- doing presorting for efficiency reason
- Each map task has circular memory buffer and write o/p.
- 100MB (mapreduce.task.io.sort.mb)
- When threshold (80%) - mapreduce.map.sort.spill.percent background thread start spill contents to disk.
- If buffer fills - map will block until spill is complete
- By write to disk - thread divide data into partitions
- Within each partition - thread perform in memory sort
- Good idea - Compress map o/p - faster, save disk space reduce amt of data to transfer
- Def - not compressed (mapreduce.map.output.compress | mapreduce.map.output.compress.codec)

Reducer side

- map task o/p is on diff m/c, Reducer can be on another m/c
- map task may finish at diff time, so ~~start~~ reduce task start copying o/p as each completes. (copy phase)
- Reduce task - Small no of copier thread - fetch o/p in parallel (Def thread - 5)
  mapreduce.reduce.shuffle.parallelcopies

- map o/p copied to reducer task JVM memory (small) else disk.
- when in memory buffer reached threshold size | threshold of no of map o/p, it is merged and spilled to disk. and background thread merge them in large sorted files. (decompress)
- After copy sort phase - which merge the map o/p

eg 50 map o/p, merge factor = 10, 5 rounds Fig-7.5

Reduce phase - reduce func is invoke for each key in sorted o/p - o/p written to HDFS. - first block replica written to local disk.

Configuration Tuning

- Principle - Give Shuffle as much memory as possible
- Map side - best performance can be obtained by avoiding m/t spills to disk
- Reduce side - best performance is obtained when intermediate data can reside entirely in memory

Fig - Table 7.1, 7.2