CS535 BIG DATA

PART 1. BATCH COMPUTING MODELS FOR BIG DATA ANALYTICS

1. DISTRIBUTED MODEL FOR SCALABLE BATCH COMPUTING - MAPREDUCE

Sangmi Lee Pallickara
Computer Science, Colorado State University
http://www.cs.colostate.edu/~cs535

FAQs

- Programming Assignment 1
  - Due Sept. 28
  - Submission via Canvas
  - Please check the course Web Page at least twice a week

Today's topics

- YARN
- MapReduce with examples

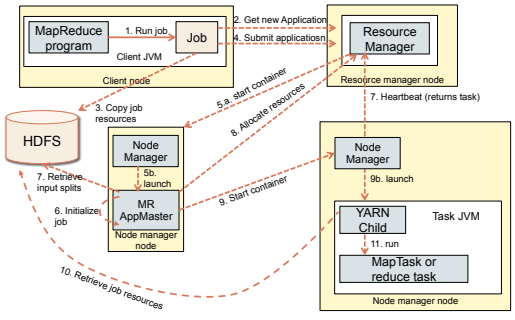
YARN (MapReduce 2)

- To provide the scalability to MapReduce
  - Splitting responsibility of the jobtracker
    - Scheduling
    - Task progress monitoring
- MapReduce is one type of YARN application

YARN (MapReduce 2)

- Resource manager
  - Manages the use of resources across the cluster
- Node manager
  - Launches and monitors the compute containers on machines in the cluster
- Application master
  - Manages the lifecycle of applications running on the cluster
  - Application master negotiates with the resource manager for cluster resources
    - Number of container and certain memory limit
    - Node managers oversee containers not to use more resources than allocated

A MapReduce job using YARN



Job Initialization

- `submitApplication()`
  - Resource manager will hands off the request to the scheduler
  - The scheduler allocates a container
    - The resource manager then launches the application master's process in the container
- Application master
  - The application master for MapReduce jobs
    - MRAppMaster
    - Initializes the job by creating bookeeping objects
      - To keep track of the job's progress
  - Retrieves the input splits
  - Creates a map task object for each split
  - Creates reduce task object
    - Mapreduce.job.reduces property

- Application master
  - Plans job execution
    - If the jobs is small, Application Master will run the tasks in **the same JVM as itself**
    - **Uber task**
      - The overhead of allocating and running tasks in new container outweighs the gain to be had it running them in parallel, compared to running them sequentially on one node

What is a small job?
A small job is one that has less than 10 mappers, only one reducer, and an input size that is less than the size of one HDFS block

Task assignment [1/2]

- Application Master requests container for all the map and reduce tasks in the job
  - From the resource manager (Step 8)
- All the requests includes information about each map tasks' **locality**
  - Host and corresponding racks that the input split resides on
- Scheduler attempts to place tasks on data-local nodes in the ideal case
  - If it is not possible, the scheduler prefers rack-local placement
    - Job is running on a node in the same rack

Task assignment [2/2]

- Requests specify required **memory**
  - 1024MB (by default)
  - This is configurable
    - `mapreduce.map.memory.mb`
    - `mapreduce.reduce.memory.mb`
- In YARN, resources are managed more fine-grained
  - Applications may request a memory capability that is anywhere between **the minimum allocation and a maximum allocation**
  - `yarn.scheduler.capacity.minimum-allocation-mb`
    - Default minimum: 1024MB
  - `yarn.scheduler.capacity.maximum-allocation-mb`
    - Default maximum: 10240 MB
  - Tasks can request any memory allocation between 1 and 10GB(default) in mulitple of 1GB
    - `mapreduce.map.memory.mb` and `mapreduce.reduce.memory.mb`

Task execution

- Application master starts the container by contacting node manager
  - The task is executed by `YarnChild`
  - `YarnChild` runs in a dedicated JVM

Progress and status updates

- Task reports its progress and status back to its application master
  - Every 3 seconds over the umbilical interface
- The client polls the application master every second
  - `mapreduce.client.progressmonitor.pollinterval`

Failures

Failures observed in Google Data Centers

"In each cluster's first year, it's typical that 1,000 individual machine failures will occur; thousands of hard drive failures will occur; one power distribution unit will fail, bringing down 500 to 1,000 machines for about 6 hours; 20 racks will fail, each time causing 40 to 80 machines to vanish from the network; 5 racks will "go wonky," with half their network packets missing in action; and the cluster will have to be rewired once, affecting 5 percent of the machines at any given moment over a 2-day span. And there's about a 50 percent chance that the cluster will overheat, taking down most of the servers in less than 5 minutes and taking 1 to 2 days to recover."

http://www.datacenterknowledge.com/archives/2008/05/30/failure-rates-in-google-data-centers/

Failures in Classic MR

- The child task fails
  - Runtime exception from the user code
  - The child JVM reports the error back to its parent before it exits
    - Written in the user logs
  - `Tasktracker` marks the task attempt as failed
    - Frees a slot to run another task
  - Sudden exit of the child JVM
    - `Tasktracker` notices that the process has exited and mars the attempt as failed
- Hanging tasks
  - If there is no progress update for a while
    - Mark the task as failed
    - Timeout period is normally 10 minutes
      - `mapred.task.timeout`

`Tasktracker` failure in Classic MR

- `Tasktracker` stops sending heartbeats
  - `Jobtracker` will notice if it hasn't received one for 10 minutes (configurable)
  - Remove it from the pool of `tasktracker`
- `Jobtracker` arranges tasks including the completed jobs
  - Because the output may not be accessible
- `Tasktracker` can also be blacklisted if more than four tasks from the same job fail (set by `mapred.max.tracker.failures`)
  - Blocklisted `tasktrackers` are not assigned tasks.
    - Until faults expire

`Jobtracker` failure in Classic MR

- The most serious failure mode
- Hadoop has **no mechanism** for dealing with `jobtracker` failure
  - Single point of failure
- **All running jobs fail**
- After restarting a `jobtracker`
  - Job should be **resubmitted**
- **This is improved with YARN**

Task failure in YARN

- Failure of the running task is similar to the classic case
  - Runtime exception and sudden exit of the JVM are propagated back to the `application master`
  - The task attempt is marked as failed
  - Hanging tasks are noticed by the application manager by the absence of a ping over the umbilical channel

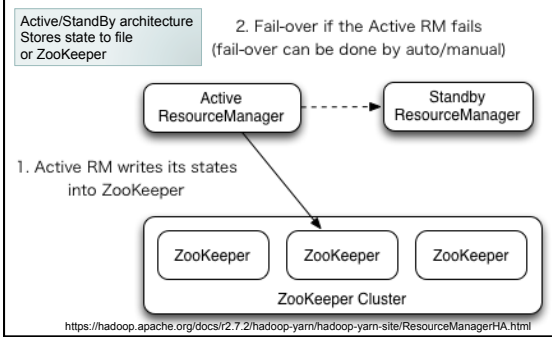
Application master failure in YARN

- No heartbeats to the `resource manager` from the `application master`
  - The `resource manager` will detect the failure and start a new instance of the `application master` running in a new container
  - All tasks will be rerun (default)
    - Recovery can be enabled
- Client will access `resource manager` to get the new address of the `application master`

Node manager failure in YARN

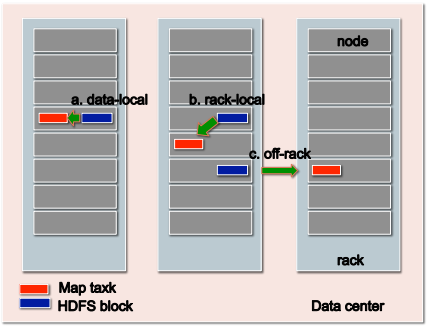
- `Resource manager` will stop getting heartbeats
  - Remove the failed node manager from the pool of available nodes

Resource Manager fails in YARN



https://hadoop.apache.org/docs/r2.7.2/hadoop-yarn/hadoop-yarn-site/ResourceManagerHA.html

Data movement in Map tasks



Data locality optimization

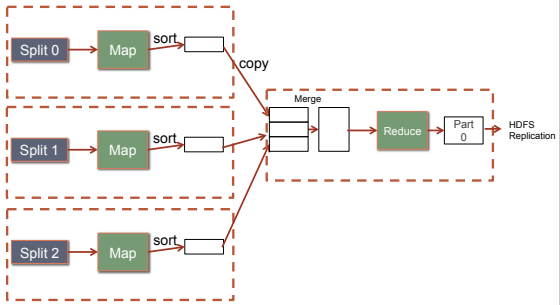
- Hadoop tries to run the map task on a node where the input data resides in HDFS
  - Minimizes usage of cluster bandwidth
- If all replication nodes are running other map tasks
  - The job scheduler will look for a free map slot on a node in the same rack

Shuffle and Combiner

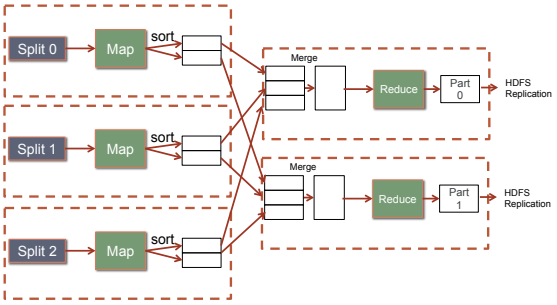
Shuffle

- The process by which the system performs the sort and transfers the map outputs to the reducers as inputs
  - MapReduce guarantees that the input to every reducer is sorted by key

MapReduce data flow with a single reducer



MapReduce data flow with multiple reducers



Combiner functions

- Minimize data transferred between map and reduce tasks
- Users can specify a **combiner function**
  - To be run on the map output
  - To replace the map output with the combiner output

Combiner example

- Example (from the previous NCDC max temp example)
  - The first map produced,
    - `(1950, 0), (1950, 20), (1950, 10)`
  - The second map produced,
    - `(1950, 25), (1950, 15)`
  - The reduce function is called with a list of all the values,
    - `(1950, [0, 20, 10, 25, 15])`
  - Output will be,
    - `(1950, 25)`
- We may express the function as,
  - `max(0, 20, 10, 25, 15)`
    `= max(max(0, 20, 10), max(25, 15))`
    `= max(20, 25) = 25`

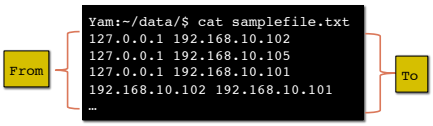
Combiner function

- Run a **local** reducer over Map output
- Reduce the amount of data shuffled between the mappers and the reducers
- Combiner cannot replace the reduce function
  - **Why?**

MapReduce Programming with Examples

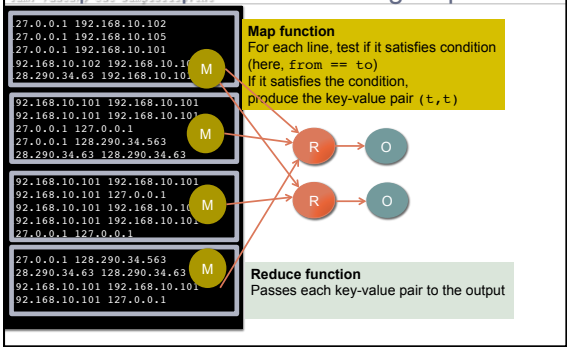
Example-1: Selecting rows

- Suppose that you have a fileset with a 1TB dataset



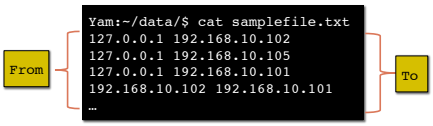
- **Retrieve all of the rows where "`from`" and "`to`" are the same URL**
  - Equivalent to SQL's "`select * where from == to`" operation

Example-1: Implementation using MapReduce



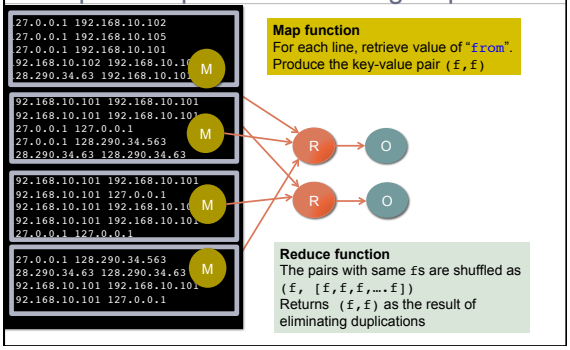
Example-2: Selecting columns

- Suppose that you have a file set with 1TB dataset

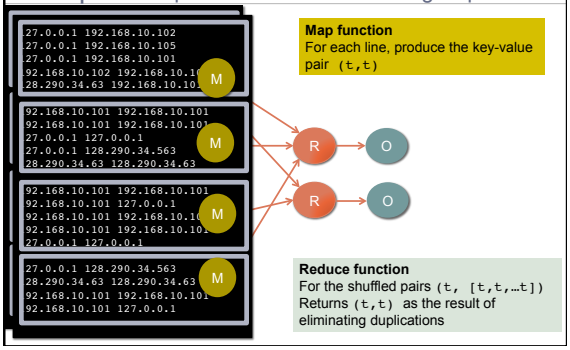


- Retrieve all ***unique*** "`from`"s

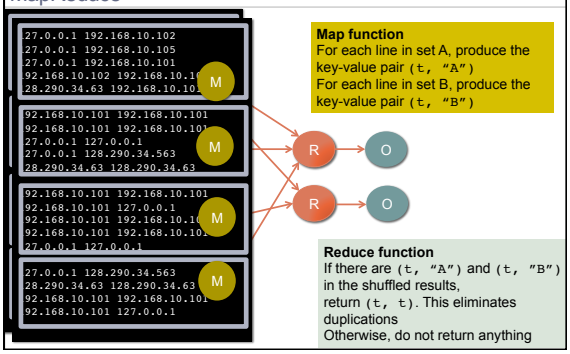
Example-2: Implementation using MapReduce



Example-3: Implementation of `Union` using MapReduce



Example-4: Implementation of `Intersection` using MapReduce



Map function
For each line in set A, produce the key-value pair `(t, "A")`
For each line in set B, produce the key-value pair `(t, "B")`

Reduce function
If there are `(t, "A")` and `(t, "B")` in the shuffled results, return `(t, t)`. This eliminates duplications
Otherwise, do not return anything

Example-5: Natural Join

- Suppose that you have two tables with 1TB each

```
Yam:~/data/$ cat samplefile.txt
127.0.0.1 192.168.10.102
127.0.0.1 192.168.10.105
127.0.0.1 192.168.10.101
192.168.10.102 192.168.10.101
…
```

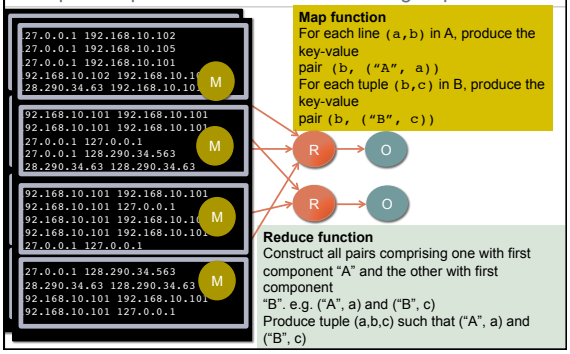
A.txt

```
Yam:~/data/$ cat samplefile-detailed.txt
192.168.10.102 ALIVE
192.168.10.101 DOWM
192.168.10.101 ALIVE
127.0.0.1 DOWN
…
```

B.txt

- If the tuples agree on an attribute that are common to the two schemas, then produce a new tuple that has components for each of the attributes in either schema
  - Equivalent to SQL's "`join`" operation
  - second url in the `A.txt` and first attribute in `B.txt` are common
  - `join(A , B) ={(url1, url2, ALIVE), (url2, url3, DOWN)}`

Example-5: Implementation of Natural Join using MapReduce



Map function
For each line `(a,b)` in A, produce the key-value pair `(b, ("A", a))`
For each tuple `(b,c)` in B, produce the key-value pair `(b, ("B", c))`

Reduce function
Construct all pairs comprising one with first component "A" and the other with first component "B". e.g. ("A", a) and ("B", c)
Produce tuple (a,b,c) such that ("A", a) and ("B", c)

Matrix-Vector Multiplication Using MapReduce

Matrix-Vector multiplication using MapReduce (1/3)

- Suppose we have an n x n matrix M, whose element in row i and column j will be denoted M_{ij}
- v is a n x 1 column vector
- Then the matrix-vector product is the vector x of length n, whose i^{th} element x_i is given by:

$$x_i = \sum_{j=1}^{n} m_{ij} v_j$$