MapReduce

Sunday, April 19, 2020 5:22 PM

Map:

- Process individual records to generate intermediate key/value pairs

Ex.

Input: "Welcome Everyone
Hello everyone" =>

Key	value	
welcome	1	Map Task 1
Everyone	1	Map Task 1
Hello	1	Map Task 2
Everyone	1	Map Task 2

↳ Parallely, process individual records to generate intermediate key value pairs

- very large data sets can be "sharded", i.e split into multiple map tasks,

Reduce:

- reduce processes and merges all intermediate values associated per key:

Key	value
welcome	1
Everyone	1
Hello	1
Everyone	1

=>

Key	value
Everyone	2
Hello	1
Welcome	1

- each key is assigned to one reduce task

- parallely processes and merges all intermediate values by partitioning keys

Key	value
welcome	1
Everyone	1
Hello	1
Everyone	1

Reduce Task 1 →

Key	value
Everyone	2
Welcome	1

Reduce Task 2 →

Key	value
Hello	1

- Hash partitioning is a popular method for task assignment
  - ↳ key is assigned to reduce # = hash(key) % number of reduce servers
  - ↳ hashing leads to fairly good load balancing in the system.

Examples

- Distributed Grep:
  - input: large set of files
  - output: lines that match pattern

MAP → emits a line if it matches the supplied pattern

REDUCE → copies the intermediate data to the output

- Reverse Web-Link Graph
  - input: Web graph: tuples (a, b) where page a → page b
  - output: for each page, list of pages that link to it

target = Key

MAP → process web log and for each input <source, target>, it outputs <target, source>

REDUCE → emits <target, list(source)>

- Count of URL access frequency
  - input: log of accessed URL's eg from a proxy server
  - for each URL, % of total accesses for that URL

MAP → process web log and outputs <URL, 1>

REDUCE → Emits <URL, URL_count>

MAP → processes <URL, URL_count> and outputs <1, (<URL, URL_count>)>

REDUCE → sums up URL_count to get total count and then emits multiple <URL, URL_count/overall_count>

MapReduce Scheduling:

Programming MapReduce:

Externally: For User:

1) write a Map program (short), write a Reduce program (short)
2) Submit job; wait for result
3) Need to know nothing about distributed/parallel computing

Internally: For the paradigm and scheduler

1) Parallelize Map
2) Transfer data from Map to Reduce
3) Parallelize Reduce
4) Implement storage for Map input, Map output, Reduce Input, and Reduce Output
5) It also ensures the barrier between the Map phase and the Reduce phase.
   ↳ no reduce can begin until all Maps complete (in general)

Inside MapReduce:

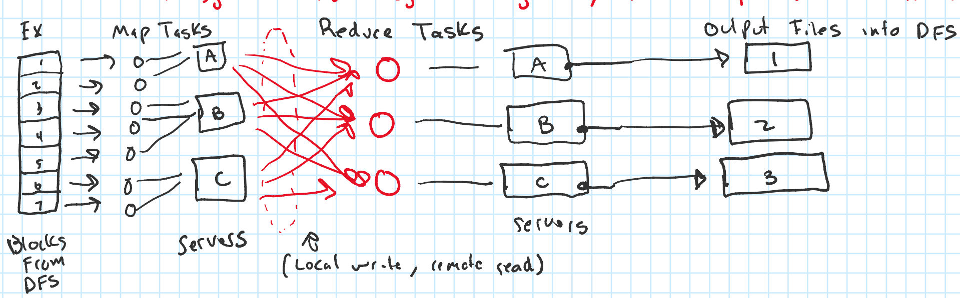
1) Parallelize Map: easy! each map task is independent of the other!
   - all map output records with same key assigned to same Reduce.

2) Transfer data from Map to Reduce:
   - all map output records with same key are assigned to the same reduce task
   - we can use a partitioning function like hash(key) % number of reducers

3) Parallelize Reduce: easy! each reduce task is independent of one another!

4) Implement storage for Map input, Map Output, Reduce Input, and Reduce Output
   - Map input: from distributed file system
   - Map output: to local disk (at Map Node) - uses local file system
   - Reduce input: from (multiple) remote disks, uses local file systems
   - Reduce output: to distributed file system

- local file systems = Linux FS, etc
- distributed file systems: GFS (Google File System), HDFS (Hadoop Distributed File System)



★ Resource Manager (assigns maps to reduce servers)

★ Each map task may produce key value pairs for any of the reduce tasks

Yarn Scheduler:

- Used in Hadoop 2.x+
- Yet Another Resource Negotiator
- Treats each server as a collection of containers
  - container = some CPU + some memory

- Has 3 components
  - Global Resource Manager (RM)
    - scheduling
  - Per-server Node Manager (NM)
  - Per-application (job) Application Master (AM)
    - container negotiation with RM and NMs
    - Detecting task failures of that job

Yarn: How a job gets a container

- the application master of some node tells the resource manager that it needs a container. The task is queued.
- some node who's task is finished tells the resource manager a container is available
- the resource manager informs the requesting node that a container is available at some node X
- the requesting node requests node X to complete its task.

Fault Tolerance:

- most common failure is a server failure
  - NM sends "heartbeats" to the RM
    - RM lets all affected AM's know and AM's take action
  - NM keeps a track of each task running on its server
    - if task fails while in-progress, mark the task as idle and restart it
  - AM heartbeats to RM
    - on failure, RM restarts AM, which then syncs up with it's running tasks.
- RM Failure
  - use old checkpoints and bring up secondary RM
- Heartbeats also used to piggyback container requests to avoid extra messages