Chapter 10 Batch Processing

Book Summary - Designing Data-Intensive Applications

Types of Systems
1. Online Systems — [services] — process request as soon as it's received and send a response
2. Offline Systems — [batch processing systems] — takes large amount of data —> process —> generate output
3. Near-real time systems — [stream processing systems] between online and offline — there might just be a ack response and output sent via events — Chapter 11

Processing data with Unix tools

Example — top 5 popular pages on your website

cat /var/log/nginx/access.log = read the log file

Awk '{print $7}' = split each line into whitespace and output 7th field
Sort = sort alphabetically
Uniq -c = removes duplicate , -c adds a counter
Sort -r -n = sort by number of times page accessed
Head -n 5 = output first 5

Sorting vs in-memory aggregation
1. If dataset has lot of duplicates, it can fit into in-memory hash table
2. Else we can use something similar to SSTables and LSM Trees - combination of in-memory and disk.
3. Unix utility 'sort' automatically spills data to disk and parallelise sorting across multiple cpu cores

Unix Philosophy
1. Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new "features".
2. Expect the output of every program to become the input to another, as yet unknown, program. Don't clutter output with extraneous information. Avoid stringently columnar or binary input formats. Don't insist on interactive input.
3. Design and build software, even operating systems, to be tried early, ideally within weeks. Don't hesitate to throw away the clumsy parts and rebuild them.
4. Use tools in preference to unskilled help to lighten a programming task, even if you have to detour to build the tools and expect to throw some of them out after you've finished using them.

Limitation of unix tools — they run on single machine.

MapReduce and Distributed FileSystems

MapReduce job —
* takes one or more inputs and produces one or more outputs
* Read and write files on distributed file system — HDFS

HDFS —
* based on shared-nothing principle (No special hardware is needed, only computers connected by datacenter network)
* A daemon runs on each node, exposing network service for other node to access files on that node.
* NameNode — keep track of which file blocks are stored on which node
* For failures — replication to multiple nodes

MapReduce Job Execution —

Mapper
1. Take the input record
2. Extract key and value
3. Write output to HDFS

MapReduce framework
1. takes key-value pair from mapper
2. Collect values belonging to same key
3. Write output to HDFS

Reducer
1. Take collection of values as input
2. Produce output . e.g. count of characters

General Details —
1. Number of mappers = number of input files.
2. Number of reducers is configured by person executing the job
3. Framework use hash of the key to Ensure all same keys end up on same reducer
4. Sorting is done in stages (because large data size) similar to SSTables/LSM Trees
5. Shuffle —
    1. Maps writes output to file.

2. Schedular informs reducer work is done.
3. Reducers connect to mappers to download sorted key-value pair for their partition.

MapReduce Workflows

Multiple mapReduce jobs chained together in workflows. Output of one job is used as input to another job.

Example workflow tools — Oozie, Airflow

Reduce Side-Joins and Grouping

MapReduce job is similar to full table scan — read entire dataset. Example — two datasets

User 101 clicked button
User 105 accessed page
User 101 load page
User 102 clicked on hyperlink

Log Entries - user activity

101	Msdeep14
102	Mandeep Singh
105	MsDeepSingh

User Info Table

For collecting data — infer who is user from log entries
1. Access log entry one-by-one and call database —> not efficient
2. Local Cache — Take a dump of database to HDFS where Map-Reduce is running —> Better Approach

Sort Merge Joins

MapReduce Execution
Mapper1 - extract user id (key), activity details (value) from user activity log entries
Mapper2 - extract user id (key), user details (value) from user database

Output —

* the MapReduce framework partitions mapper output by key and then sort key-value pair —> all records (user activity & user database) with same user id become adjacent
* This is used as reducer input
* Reducer can process all records for a user id in one go — keeps only one user record in memory at a time & no network calls.
* Called sort-merge join [Mapper output sorted by key; reducers merge the sorted lists of records from both sides of join]
* The join work is done by Reducers; also referred to as **Reduce-Side Joins**.

The related data is at the same place — helpful in achieving GROUP BY kind of operations.'
e.g. counting page views

What about celebrity user?
Bringing all data at single place for single user will break (creates skew/hot spots) — these records called linchpin objects —
* one reducer doing lot of work as compared to others
* Can take lot of time to MapReduce task completion
* So instead, mappers send the hot key to random reducers based on some hash — parallelisation

Map Side Joins

Mapper reads file from HDFS and writes back to HDFS — no reducers or sorting.

1. Broadcast hash joins
    1. Large dataset joining with smaller dataset (can fit into memory of each mapper)
    2. Essentially, smaller dataset is broadcasted to all mappers
    3. Pig - replicated join; Hive - MapJoin
2. Partitioned Hash Join
    1. Inputs to map-side join are partitioned in same way
    2. e.g. userId%10 — so 10 partitions
    3. Hive - bucketed map joins
3. Map-side merge joins
    1. Partitioned in same way and sorted based on same key

Output of Batch Workflows

Result of data processing?

1. Building Search Indexes
2. Key-value stores as batch process output
    1. Spam filters, anomaly detection, recommendation systems
    2. The output is saved such as it can be queried by some key. e.g. suggested connections for a user on LinkedIn

Comparing Hadoop to Distributed Databases

1. Diversity of storage
    1. Databases are more structured, Hadoop work on files- can be written using any data model or encoding
    2. Dump anything to files and later figure out how to process it - Don't have to worry about schema design
2. Diversity of processing models
    1. MapReduce allows to run your own code on large datasets and don't depend on database software.
    2. Build tools on top of MapReduce as needed. e.g. Hive
3. Designing for frequent faults
    1. If the task is aborted, it can be retried
    2. This enables better resource utilisation — allows tasks with more priority to run prior.

Beyond MapReduce

Writing Map-Reduce jobs from scratch is hard and time-consuming.

Materialisation of Intermediate State

[input] [MapReduce] [output]
[input] [MapReduce] [output]

Output of one job acts as input to another job. Writing out this intermediate state of files is called materialisation.

Issues —
* MapReduce job can only start when preceding job is completed — more execution time
* Intermediate files are replicated on HDFS — overkill for temporary files

To fix these; multiple **Dataflow engines** such as Spark, Tez, Flink — Handle an entire workflow as one job instead of breaking into multiple sub-jobs
* sorting is done only when needed
* No unnecessary map tasks
* Locality optimisations — task consumer and producer on same machine so data transfer over network
* Can write to local disk or memory instead of HDFS — no replication needed

Fault Tolerance
* MapReduce writes intermediate states to HDFS, task can be restarted on another machine on failure
* Spark don't write intermediate state to HDFS; so if machine is lost; data is recomputed

  * For this it keeps track of how data was computed — use Resilient Distributed Dataset (RDD)
* The recomputed data should be same; if not; then entire job should be re-run
* If computation is CPU intensive; materialisation of intermediate states is chapter

Materialisation — as compared to MapReduce, write only needed intermediate states and not all.

High Level APIs and Languages

Writing MapReduce jobs is laborious task - so lot of new platforms came up — hive, Pig, etc.

Writing code - requires more expertise and work
Provide declarative interfaces makes it easy — and internal execution engine can also decide the way to execute for faster task completion.
* Spark, Hive and Flink have cost-based query optimisers — that can change the order of joins and reduce the number of intermediate states.
* Instead of reading entire record from disk, read only required fields

Subscribe on YouTube — @MsDeepSingh

Read my O'Reilly book "System Design on AWS"