**Designing Data-Intensive Applications**

Chapter 10: Batch Processing

**Chapter Introduction**

* Throughout the book we have looked at request, queries, responses and results. Success of these is typically measured in response time
* In batch jobs you might not care so much about response time, but rather throughput
* 3 kinds of different systems
  + Services
  + Batch processing
  + Stream processing

**Batch Processing with Unix Tools**

* Example showing parsing a log, grabbing a URL and finding the top 5 unique
* You can do this by chaining Unix commands or via custom programs
* Custom program keeps counters in memory while unix pipeline does not, can make use of disk
* If high number of items to count then custom program will eventually run out of memory

**Unix Philosophy**

* Core philosophies
  + Make each program do one thing well
  + Expect output to become input
  + Design and build to be tried early and iterate
  + Use tools to lighten a programming task
* Unix favors a common interface stdin and stdout always ascii

**MapReduce and Distributed Filesystems**

* Similar to Unix tools but distributed across thousands of machines
* Uses HDFS (Hadoop Distributed FIle System)
  + Shared-nothing principle
* Central NameNode keeps track of where file blocks are stored
* Information replicated across machine for fault tolerance

**MapReduce Job Execution**

* Programming framework used to write code to process large datasets in distributed filesystems
  + Read a set of input files and break into records
  + Call mapper function to extract a key and value
  + Sort key-value pairs by key
  + Call reducer function to iterate over the sorted key-value pairs, sorting has put matching keys adjacent so easy to do various aggregations
* Mapper
  + Called once for every record
  + Extracts key and value
  + For each input may generate any number of key-value pairs (or none)
  + Only given one record at a time, does not maintain any state
* Reducer
  + Collects all values from the same key and iterates over that collection
  + Can produce output records like the number of occurrences of the same URL
* Typically written in conventional programming language
* Code for mapper and reducer needs to be sent to all systems before job can be run
* Shuffle?
* Common to have multiple jobs chained together, typically called a workflow. Various tools designed to manage these, Oozie, Azkaban, Luigi, Airflow, Pinball

**Reduce-Side Joins and Grouping**

* Common to have a record of information associated with another record, foreign key in relational model
* In relational DB might use an index, but mapreduce always does a full table scan which is much more expensive
  + Only really makes sense to use mapreduce when you are doing it across all users for example, not a join for a single user
* May need to associate user activity with user profile information
  + Querying an external data source might be extremely slow especially with the high number of requests put out by mapreduce
  + Best to try to promote as much locality as possible
  + Better to put a backup of the database in HDFS
* Sort-merge joins
  + MapReduce job can arrange records to be sorted such that the reducer always sees records from user db first followed by activity events in time order, called a secondary sort
  + Reducer called once for every user ID and has the first value be the date-of-birth record and then iterate over later activity outputting pairs of viewed url and viewer age in years
* Group By
  + The simplest way to implement is to have mapper emit key-value pairs to produce the desired grouping of keys
* Handling skew
  + If one machine receives all of the records for a certain ID, might end up with some machines with way more records than another
  + Hot spots caused by this skew
  + The entire job can be waiting on a single high record ID
  + Possible to fight with various different tools

**Map-Side Joins**

* If you can make assumptions about your input data it is possible to make joins faster with map-side joins
* Mapper simply reads one input file block and creates one output file
  + Broadcast hash joins
    - Large dataset + small dataset (can fit entirely in memory)
    - Load user database and then mapper can scan over user activity events and look up user ID for each event in the hash table
    - Possible to put into a read-only index on disk as well
  + Partitioned hash joins
    - Put all info you want to be joined intelligently into a single partition so it can be joined on one machine

**Output of Batch Workflows**

* Not transaction processing and not purely analytics
* For building search indexes at Google
  + Outputs new index periodically and supersedes the previously outputted one for search
* Transferring output to new database for querying
  + Not good to write records individually for various reasons, better to transfer entirely once job is done
* Similar principles to Unix, treat inputs as immutable and expect output to become input
* Easy to roll back mistakes by rerunning jobs with old code

**Comparing Hadoop to Distributed Databases**

* Hadoop more like on-read schema
* Data can be dumped in HDFS and schema applied later, does not require data to be modeled beforehand
* Designed with fault tolerance in mind because systems would lose priority for map reduce jobs at google, not necessarily because machines were unreliable

**Beyond MapReduce**

* Difficult to write map reduce jobs, various tools have been written to make it easier and abstract hairy parts away
  + Pig, Hive, Crunch, Cascading

**Materialization of Intermediate State**

* Every output from MapReduce is represented by a file, not like Unix where things are directly piped
* One job needs to finish before the next is started
* Often redundant and slow reading through the whole input file on each node again
* Dataflow engines try to mitigate some of these points in various different ways
  + Spark, Tez, Flink
* Without materialization some fault tolerance becomes more tricky, but might not be strictly necessary

**Graphs and Iterative Processing**

* Possible to store graph in a distributed filesystem but certain jobs make it a bit awkward to process with
* Many graph algorithms require repeat until convergence
  + This can be done by just iteratively running MapReduce jobs until condition is met, but might be very inefficient
* Tools like pregel try to make graph computation more efficient by simplifying and removing unnecessary parts

**High-level APIs and Languages**

* Things moving in the direction of more declarative query languages as opposed to custom written programs for map and reduce
* Trade-off between flexibility and ease of use
* Various common queries have begun being baked in like k-nearest-neighbors in certain libraries that help with MapReduce/hadoop jobs