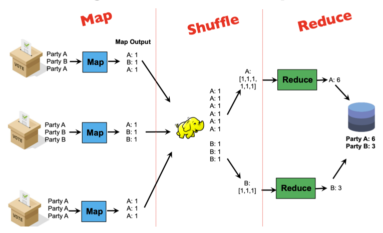
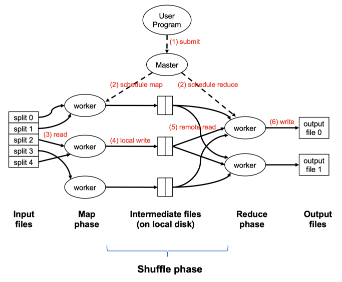
MapReduce





If size of map task too small, lead to high overhead/large amount of disk and network IO. If size too big, limited parallelism/cannot fit into memory of workers. **Map Task** - unit of work. **Mapper** - process executing map task. **Map Function** - single call to map. Barrier between map and reducer (but possible to begin copying intermediate data earlier.). Keys arrive at each reducer in sorted order but no enforced ordering **across** reducers.

Partitioner

Between shuffle and reduce phase. By default, assignment to reducers done by a hash function. Possible to write custom partitioner to further spread the values.

Combiner

Reduce the number of disk writes. Combiner does not affect correctness of output. In general, combiner operations must be commutative $(a + b = b + a)$ and associative $(a + (b + c) = (a + b) + c)$. Same input type as output of mapper and same output type as input of reducer.

Performance

Linear scalability - more nodes means more work in the same time. **Minimise network and disk IO** - reduce random/individual access and more bulk access. **Memory working set of each worker** - larger set means higher probability of out-of-memory errors.

More Concepts

Preservestate by using a hash table(reduces IO but increases memory working set.) Secondary sort - define a composite key to sort values when arrived at reducers. Partitioner needs to be customised to partition by the original key (natural key) only.

Hadoop File System

Client first approaches namenode which checks where the data is stored in datanode. Datanode checks metadata to see which name node has the data, and redirect client there. Then client goes to appropriate datanode to get data. **Replication** - Namenode decides which datanodes to be used as replicas. **No data is moved through namenode**. If data in namenode is lost, no way to reconstruct them. But hadoop has backup namenodes.

Relational DB

Star schema - keys in the middle with table around it. Perform join operation on keys table with others to obtain the key.

- Projection and Selection - value for select clause. Just use map to take in a tuple and emit with appropriate attributes/meet predicate.
- Group by - map over tuples and emit based on group by predicate. Compute aggregate function in reducer.

Relational Joins

Broadcast join - all mappers store a copy of the small table. Then iterate over big table and join records with small table. **Reduce-side join** - doesn't have memory constraint but slower than map-side (because of shuffle phase). Different mappers operate on each table, emit records with key as variable to join by. Hold keys from one table in memory and cross them with records from other tables. Primary key is the table name, secondary key is the attribute of interest.

Similarity Search

To find similar documents:

- Shingling - convert document to sets of short phrases (shingles)
- Min-hashing - convert sets to short signatures while preserving similarity. Documents with same signature are candidate pairs.

Shingling - k-shingles means a sequence of k tokens(a sliding window of size k) that appear in document. Similarity Metric for shingles can be represented as a matrix where columsn represent documents and shingles represent rows. Similarity between 2 documents measured using **Jaccard Similarity** (ratio of intersect size to union size).**Min-hashing** - Hash each shingle and take the minimum value to represent the document. Find hash function that collides when similarity is high. **Probability that 2 documents have same MinHash is equal to their Jaccard similarity**. Possible for detection to fail if min hash value isnt the same as the hash value of the common shingle. Candidate pairs are documents with the same final signature. Possible to use multiple hash functions and generate multiple signatures for each document.

MapReduce Implementation

Map - read and extract shingles. Then hash each shing and take the minimum. Emit the signature with the document id. Reduce - receive all given signatures and generate pairs, can also compare each pair to check if they are similar.

Clustering

K-means algorithm - pick K random centers. Assign each point to nearest cluster. Move each cluster to average of its assigned points. Repeat till no new assignments. MapReduce job performs one iteration of k-means. Without in-mapper combiner, emitting every point is expensive, O(nmd), where n is number of points, m is number of iterations and d is dimensionality of point. Instead use in-mapper combiner to improve time to O(kmd) where k is number of centers, by emitting centers with set of points.

NoSQL

Horizontal partitioning - more data simply partition to more shards, improves speed due to parallelisation. Duplication - allows queries to go to 1 collections. Relaxed consistency guarentees - prioritise availability over consistency - can return stale data.

Key-Value Store

Keys are usually primitives and can be queried (ints, strings). Values can be primitive or complex;usually cannot be queried. Non-persistent implementation is jus a hash-table in memory. Persistent implementation is stored persistently to disk.

Document Stores

Database can have multiple collections and collections have multiple documents and each document is like a JSON object. Format of documents not enforced. Queries can be CRUD

Wide Column Stores

Related groups of columns are grouped as column families. Sparsity: if a column is not used for a row, it doesn't use space.

Concepts

Eventual consistency: if the system is functioning and we wait long enough, eventually all reads will return the last written value. Strong consistency: any reads immediately after an update must give the same result on all observers. Basically Available: basic reading and writing operations available most of the time. Soft State: without guarentees, we only have some probability of knowing the state at any time

Eventual consistency offers better availability at the cost of a much weaker consistency guarentee.

Joins

Tables designed around queries expected to retrieve. Duplicate the tables then perform the joins since disk is cheap. However, updates to primary tables needs to be propagated to multiple tables.

Concepts of Distributed Databases

Assumptions: all nodes in distributed database are well-behaved, users should not be required to know how data is physically distributed/partitioned/replicated, query that works on a single node database should work on a distributed database. Sharing everything (single node), sharing memory (supercomputers), sharing disk(cloud databases), shared nothing (NoSQL).

Horizontal Partitioning

Put different tuples into different nodes (sharding). Partition key is variable used to decide which node tuple will be stored on —(same shard key means same node). Good if mostly aggregate data only within key. Bad is value of key is very imbalance, causing a lack of scalability. **Range Partition** - split partition key based on range of values. Can lead to imbalance shards. Splitting the range is automatically handled by balancer. **Partition Pruning** - searching for particular value is easier and faster.**Hash Partition** - hash partition key then divide into partitions based on range. Possible to add/remove node using consistent hashing.

Consistent Hashing

Imagine output of hash functions lies on a circle. Each node would act as a marker. Each tuple is placed on the circle and assigned to the node that comes clockwise-after it. To delete, re-assign all its tuple to node clockwise after this. Similarly, nodes can be added by splitting the largest current node into 2. **Replication** - replicate in the next few additional nodes clockwise after primary use to store it. **Multiple markers** - multiple markers per node. For each tuple, assign it to the marker nearest to it in the clockwise direction. When removing a node, tuples will not all be reassigned to the same node; load balancing is better.

Query Processing

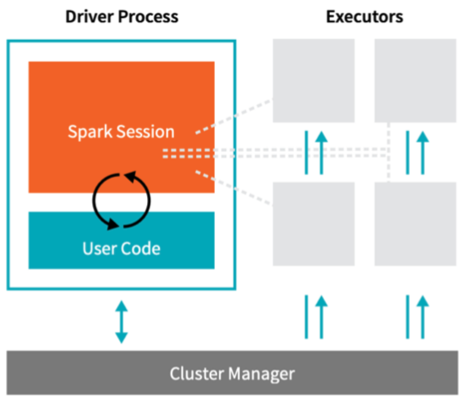
Routers - handle requests from application and route queries to correct shards. Config server - store metadata about which data is on which shard. Shards - store data and run queries on their data. App makes query to router. With config server, routers determine which shards should be queried. Query sent to relevant shard. Shards run query on their data and return result back to routers. Routers merge results and return return merged results.

Spark

Does computations in memory. When memory is insufficient, spill to disk. Easier to program than Hadoop.

Architecture

Spark uses a cluster manager to manage resources(memory). The Driver Process responds to user input, manages Spark application and distributes work to Executors which runs the code assigned to them and send results back to driver process. In local mode, all process run on the same machine.



RDDs

RDDs are **immutable**. RDD partitioned into multiple workers. **Transformations** transforms RDDs to RDDs by creating new ones. Transformations are lazy to optimise for speed, only triggered when action is called. Examples include *map*().**Actions** trigger Spark to compute result from transformations, and sends results to driver node. *collect*() asks Spark to retrieve all elements of RDD to driver node. Since RDD are distributed across machines, transformations executed in parallel, results sent to driver only in final step. RDD can also be in the disk if out of memory. *sc.textFile*() reeads file on each worker node in parallel, not on driver node.

Caching, DAGs

When its expensive to perform repeated computations, just cache the results using *cache*() to memory of worker node. Caching should be done when its expensive to compute and needs to be re-used multiple times. When worker has no more memory, evict least recently used result. If not enough memory to cahce an RDD, it will be ignored. Spark creates a DAG to represent transformation of RDDs. Transformation create graph, Actions trigger computations. Narrow dependencies are where each partition of parent is used by at most 1 partition of child. Wide dependency means each parent used by multiple partitios of child. Consecutive narrow dependencies are grouped as stages. Within stages, Spark performs consecutive transformations. Across stages (wide dependencies), data is shuffled (similar to mapreduce which writes to disk). Minimise shuffling to improve performance.

Lineage, Fault Tolerance

Fault tolerance resolved not by duplication because memory is expensive and limited. Spark tries to store all data in memory and not disk. If a worker node goes down, replace it by a new worker node and use the graph to recompute the data in the lost partition. Only need to recomptue the RDDs from the lost partition.

Dataframes and DataSets

Dataframes represent a table of data, higher level than RDD, but still compiled to RDD. Transformation done using SQL queries which returns a Dataframe. Datasets are type-safe dataframes. Supported only languages that are not dynamically typed. Dataframe is a row in the Datasets.

Supervised Machine Learning

Classification is categorising samples into classes. Regression is predicting numeric labels. Typical ML pipeline is preprocessing $\rightarrow$ training $\rightarrow$ testing $\rightarrow$ testing $\rightarrow$ evaluation.

Preprocessing

For missing values, possibly because information not collected, or missing at random. If data not missing at random then missingness itself may be important information. To handle, either eliminate rows with missing values, fill in missing values (with mean/median, fit a model then use as training data, dummy variables, 0 or 1, to indicate whether its missing). One Hot Encoding - Convert discrete feature to a series of binary features. Clipping normalisation (rounding values), log transform (log the values), standard scalar (remove the mean), max min (0 to 1).

Classification

Sigmoid function $\frac{1}{1+e^{-x}}$ maps real numbers to the range (0,1). To predict, $\sigma(x \cdot w + b)$ to get value, w is weight and b is bias. To train, aim to minimise some loss function (in this case cross entropy loss) by using gradient descent. Predicted label referred to as $\hat{y}$ while actual is y. **Decision Trees** - classifcation with response. To test just run through the tree to obtain the final result. Decision trees are simple interpretable and fast but suffer from poor accuracy and isn't robust to small changes. Random Forests and Gradient Boosted Trees combines a large number of decision trees. On tabular data, accuracy is very good and are faster, easier to tune and more interpretable.

Evaluation

Binary classfication (either true or false). Confusion Matrix that maps predicted label to ground truth label. Accuracy is the fraction of correct predictions, $\frac{TP+TN}{TP+TN+FP+FN}$. Sensitivity fraction of positive cases detected, $\frac{TP}{TP+FN}$, specificity fraction of actual negatives correctly identified $\frac{TN}{TN+FP}$.

Pipelines

Building complex pipeline out of simple building blocks for better code resuse and easier to perform cross validation and hyperparameter tuning. **Transformers** map dataframes to dataframes. Output new dataframe which append result to original. Fitted model is a Transformer that transform dataframes in to 1 with predictions appended. Similarly a fitted model is a Transformer that transforms a DataFrame into one with the predictions appended. **Estimator** is an algorithm which takes in data and output fitted model(transformer). A pipeline chains together multiple Transformers and Estimators to form an ML workflow. Pipelines are estimators (output fited model). Training time - When $fit()$ called, start from beginning. Transformers call $transform()$ and Estimators call $fit()$ to fit data and then $transform()$ on fitted model. Test time - output of $fit()$ is estimated pipline model. Its a transformer and consists of a series of transformers. When its $transform()$ is called, each stage $transform()$ is called.

Graphs

Nodes represent objects and edges represent relationships. Think like a vertex - algorithms are implemented from the view of a single vertex, performing one iteration based on messages from its neighbors. *compute*() describes behavior at 1 vertex in 1 step.

Simplified PageRank

Page importance measured as the number of in-links but malicious users can create huge number of dummy web pages to link to their page and drive up the ranks. Instead, make number of votes proportional to its own importance. **Importance** of r_j with n outlinks then each link gets $\frac{r_j}{n}$ votes. Page's own importance is the sum of the votes on its in-links. Problems with page rank - measures generic popularity of a page (doesnt consider popularity based on specific topics), uses a single measure of importance (other models of importance), susceptible to link spam (artificial link topographies created in order to boost page rank).

Flow Model

Flow equations are equations that calculates the flow of a page. May not be solvable but additional constraint that all ranks sum to 1 enforces uniqueness. Column stochastic(columns sum to 1) matrix. Matrix entry is column(source) to row(dest). So $r_j = \sum_{i \rightarrow j} \frac{r_i}{d_i}$ this means that $i \rightarrow j$. To update the values, can be wrtten as $r = M \cdot r$. Flow equations can be solved using power iterations. Each node starts with equal importance of $\frac{1}{n}$. During each step, each node passes current importance to its neighbors. Repeat till the difference between old and new values are smaller than some constant. Only work if the update is done to all nodes at the same iteration.

Random Walk

Random surfer that starts on a random page and travels on a random outgoing link, repeats indefinitely. **Stationery Distribution** - As $t \rightarrow \infty$, probability distribution approach a steady state. $p(t)$ is probability distribution over pages at time t.Then $p(t+1) = M \cdot p(t)$.

Teleports

May not converge (when there are just 2 nodes with links to each other). Deadends (no out-links), leads to importance leaking out (converging to 0). Spider traps (out links within group), absorbs all importance. Importance in spider trap split evenly amongst nodes in trap. At each step, surfer can teleport to a random node. For deadends, make sure that matrix is column stochastic; confirm to teleport out of deadends and equally likely to end up anywhere else. $r_j = \sum_{i \rightarrow j} \frac{r_i}{d_i} + (1-\beta)\frac{1}{N}$. Teleport term is matrix of size N with all entries $\frac{1}{N}$.

Topic-Specific

$$f(x) = \begin{cases} \beta M_{ij} + \frac{1-\beta}{|S|}, & \text{if } i \in S \\ \beta M_{ij} & \text{otherwise} \end{cases}$$

Basically at any page, its possible to teleport from j to i only if i is in S. If a node is in S, its flow equation would include the teleport term value of importance as well. When random walker teleports, its picks a page from a set S. S contains only pages that are relevant to the topic. For each teleport set S, we get a different vector r_s.

Pregel

Computation consist of supersteps which is a user-defined function for each vertex that can be invoked in parallel. At a single vertex, v at superstep s, *compute*() can read messages sent to v at $s-1$, send messages to other vertices that will be read in $s+1$, or read/write values of v and value of outgoing edge(add/remove edges). Termination happens when a vertex choose to deactivate itself, woken up if new message received, and stops when all vertex are inactive.

Implementation

Master and workers architecture. Vertices hashed and assigned to workers. Worker maintain state of its portion in **memory**. Computations happen in memory. In each superstep, each worker loop through vertices and execute *compute*(). Messages sent to vertices on same worker or to vertices on different workers (buffered locally and sent as a batch). Fault tolerance is resolved by heartbeats. Corrupt workers reassigned and reloaded from checkpoints. Checkpointing (periodically saves computations to HDFS so that when failure happens, computations roll back to latest checkpoints) to persistent storage.

Giraph

Employ **Hadoop Map** to run computations. **ZooKeeper** (service that provides distributed synchronisation) to enforce barrier waits. For pagerank, in each superstep, invoke *compute*(), aggregate messages from neighbors to compute updated value then send message to neighbors. Node's current values computed by summing messages from the previous Superstep and computing the PageRank update equation below. Outgoing messages are computed by dividing each node's value equally among its outgoing neighbors.

Streams

Data arriving overtime at a rapid rate from input ports. Elements of stream are tuples. Potentially infinite, system cannot store entire stream due to limited memory.

General Stream Processing Model

Limited working set (memory), Archival storage(write info to disk), Standing queries(queries that are always present), Ad-hoc queries (unpredictable queries).

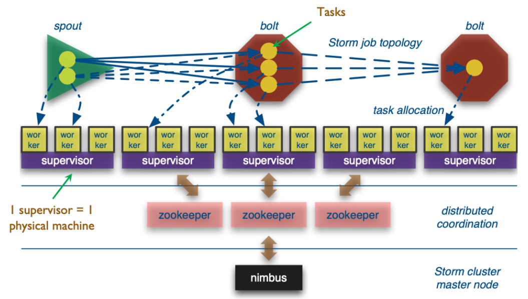
Algorithms

Reservoir sampling. Maintain a uniform random sample (each item has same probability of being sampled) of fixed size B over time. Allows estimation of almost any statistics of the data distribution. Intution - fill up reservoir first. Once filled, with probability $\frac{B}{t}$, replace item in reservoir with current item at time t.

Storm

Tuple: an ordered list of named values. User creates a topologyy which is a computation graph. Nodes hold processing logic, edges indicate communication between nodes. Each topology is a Storm **job**. Runs forever till killed. Streams: unbounded sequence of tuples, transformed by processing elements of topology. Spouts: generate streams, can propogate to multiple consumers. Bolts: subscribe to streams, process incoming streams and produce new ones, stream transformers.

Architecture



Bolts are executed by multiple executors in parallel called tasks. When bolt emits a tuple, which task should it go in the next bolt? Shuffle grouping: tuples randomly distributed across tasks. Field grouping: based on the value of user-specified field. All: replicate tuple across all tasks of target bolt.