|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Intro | | A diagram of a software application  Description automatically generated with medium confidenceChallenges of big data: Volume, Variety, Velocity, Veracity (data quality issues)  Cloud computing/utility computing (EC2, Google compute)  VM has high overhead as each app has its own OS  Containers enables lightweight sharing of resources, as apps run in an isolated way but still share same OS  Infrastructure as a service (IAAS): Utility computing  Platform as a service (PAAS): provide hosting for web apps, takes care of hardware maintenance, upgrades... (Google app engine)  Software as a service (SAAS): Gmail, Zoom. Key idea: Datacenter is the computer | | | | | | |
|  | | A graph of a line graph  Description automatically generated with medium confidenceBlade server -> Racks -> Cluster -> Data centre.  Storage capacity (lowest to highest): DRAM (dynamic RAM/memory), Flash (USB), Disk. Speed (lowest to highest): Disk, Flash, DRAM  Higher capacity = lower speed (tradeoff) (same for server, rack, cluster w increasing capacity)  Bandwidth: max amt of data that can be transmitted per unit time (GB/s)  Latency: time taken for 1 packet to go from src to dst (one-way) or src to dst to src (round trip) (in ms)  Throughput: rate of data *actually* transmitted  When moving large data, bandwidth determines time. When moving small data, latency determines time.  Disk reads higher latency, lower bandwidth than DRAM. From local to Rack to Datacenter, capacity & latency incr while bandwidth decr | | | | | | |
| MapReduce | | | | | Work -> Partition into smaller work tasks -> Assign to worker -> Results from all worker -> Combine into single result  Challenge 1: Machine failure (with more servers/machines, chances of multiple servers failing is higher, and should be expected)  A diagram of a map  Description automatically generated2: Synchronization (dk order which machine complete/ how to communicate partial results = need control mechanisms like barriers)  3: Programming difficulty (concurrency diff to program at the scale of datacentre, multiple interacting services)  Iterate over large num of records -> extract something of interest (map) -> shuffle and sort intermediate results -> aggregate intermediate results (reduce) -> generate final output  Progammers only code:  1) map (k1, v1) -> List(k2, v2) (can output multiple key value pairs)  2) reduce (k2, List(v­2)) -> List(k3, v3)  The execution framework handles all the other challenges (scheduling, data distribution, synchronization, errors and faults) | | | |
| MapReduce Implemen-tation | | | | | A diagram of a program  Description automatically generated1) Submit: user submits MapReduce program (code for map and reduce fns) & configuration (e.g. num of workers) to Master node  2) Schedule: Master schedules resources for map and reduce tasks (Master don't handle any actual data)  3) Read: Input files are separated into "splits" of ≈ 128MB each. Each split = 1 map task. Workers execute map tasks 1 at a time  - Map phase: Each worker iteratates over each <key, value> tuple in its input split, and computes the map fn on each tuple  4) Local write: Each worker writes outputs of map fn to intermediate files on its own local disk. These files are partitioned by key (i.e. data for a single key is in 1 partition, 1 rectangle on chart)  5) Remote read: Each reduce worker is responsible for 1 or more keys. For each such key, it reads the data it needs from the corresponding partition of every mapper's local disk  - Reduce phase: After getting all needed key-value pairs, computs reduce fn  6) Write: Output of reduce fn is written (usually to HDFS)  \*Shuffle phase = local write and remote read steps.  \* If each split/chunk is too big: limited parallelism. Too small: high overhead (master node is overwhelmed w scheduling work)  \* Barrier btw map and reduce phases. (Shuffle phase can still begin copying intermediate data earlier)  \* If reduce task handles multiple keys, it will process these keys in sorted order | | | |
| Key Terms | | | | | Worker = component of cluster that performs storage and processing tasks  Map Task = basic unit of work, typically 128MB. A map task is a job requiring to process 1 split; not a worker  - A single worker can handle multiple map tasks (assigned another map task once it completes 1)  Mapper or Reducer = process executing map or reduce task, not to physical machines / workers.  Map function = single call to the user defined map (k1, v1) -> List(k2, v2)  - For chart above, 5 map tasks, 5 mappers, 3 workers (map phase). - 1 map task can involves many calls to a map fn | | | |
| Writing MapReduce Programs | | | | | Programmers specify: 1) map (k1, v1) -> List(k2, v2). 2) reduce (k2, List(v­2)) -> List(k3, v3)  Programmers can optionally specify partition & combiner fns (meant to reduce network traffic) | | | |
| In voting e.g., key A went to reducer 1, key B went to reducer 2. By default, key k goes to reducer: hash(k) % num\_reducers  i.e., Hash function(key) mod num\_of\_reducers.  If we know some keys have more values/are more popular, can implement custom partition to better spread out load among reducers | | | |
| In voting e.g., writing map output to disk is expensive. Combiners locally aggregate output from mappers. (Combiners = "mini-reducers")  Map output -> Combiner output: [A: 2, B: 1]. [A: 1, B: 2]. [A: 3]  - User's responsibility to ensure combiner don't affect correctness of final output, whether combiner runs 0, 1, or multiple times  E.g. sum(sum(1,1), 1, sum(1, 1, 1)) = sum(1, 1, 1, 1, 1, 1) = 6. So summing, finding max or min would work  But mean or minus won't work: mean(mean(1, 1), 2) = 1.5 ≠ 1.3333 = mean(1, 1, 2)  Only correct to use reducers as combiners if the reduction involves a binary operation (e.g. +) that is both associative (a + (b + c) = (a + b) + c) & commutative (a + b = b + a)  Note combiner must have same input and output key-value types, which also must be the same as the mapper output type and reducer input type. This is to ensure if combiner don't run, reducer will still work | | | |
| Basic algo design | | | | Linear scalability: more nodes can do more work in the same time. - Linear on data size. - Linear on computer resources  Minimize disk and network I/O: - Min disk I/O; sequential vs random access of data. - Min network I/O: send data in bulk vs in small chunks  Reduce memory working set of each task/worker: - Working set = protion of memory that is actively being used during algo execution.  - Large working set -> high memory requirements -> prob of out-of-memory errors | | | | |
| E.g. | |  |  |  | | --- | --- | --- | | A white paper with black text and black text  Description automatically generated | A computer code with text and numbers  Description automatically generated | A computer code with text  Description automatically generated with medium confidence | | Mapper process each word 1 by 1, and emits a "1", to be summed by reducers | Mapper uses a hash table to record words and counts per line (i.e. in each call to map fn). After processing each line, it emits counts for this line | Mapper use hash table to record words and counts across all lines in a single split  By aggregating tuples across map tasks, this reduces disk and memory I/O. However, this increase memory working set | | | | | | | | |
| Secondary Sort | | | | A screen shot of a computer code  Description automatically generatedEach reducer's values arrive unsorted. If we want them to be sorted: define a new 'composite key' as (K1, K2) where K1 is the original key/natural key and K2 is the additional variable we want to sort by  - Partitioner have to be customized, to partition by K1 only, not (K1, K2). (We still want partitioner to work the same as before, use K1 to decide which tasks for reducer. Sorting only for all the K1 data that arrive at the reducer, then sort by K2)  A screenshot of a computer code  Description automatically generated  Partitioner only partition by yearMonth (K1)  Compare by K1 (yearMonth) first; if tie, then compare by K2 (temp)  Hadoop always secondary sort before giving to reducer  In contrast, if we sort data in the reducers, may be expensive (in memory & time) if there is a lot of values for 1 key. Seconday sort uses the inbuilt distributed sorting process of Hadoop to sort values. | | | | |
| Hadoop File System | | | Distributed File System: don't move data to workers; move workers to data. - Store data on local disks of nodes in cluster. - Start up workers on the node that has the data local  GFS (Google File System) for Google's MapReduce. HDFS (Hadoop Distributed File System) for Hadoop  Assumptions: 1) Commodity hardward instead of "exotic" hardware: Scale out, not up  2) High component failure rates. 3) "Modest" number of huge files (instead of many small files).  4) Files are write-once, mostly appended to. 5) Large streaming reads instead of random access (high sustained throughput over low latency) | | | | | |
| Design decisions: 1) Files stored as chunks: Fixed size (64MB for GFS, 128MB for HDFS)  2) Reliability through replication: Each chunk replicated across 3+ chunkservers  3) Single master to coordinate access, keep metadata: Simple centralized management | | | | | |
| A diagram of a computer  Description automatically generatedHDFS Architecture  - To perform replication when writing data: Namenode decides which datanodes to be used as replicas. 1st datanode forwards data blocks to 1st replica, which forwards them to 2nd replica, ...  Namenode Responsibilities:  1) Manage file system namespace:  - Holds file/directory structure, metadata, file-to-block mapping, access permissions, ...  - Coordinating file operations by directing clients to datanodes for reads and writes  - No data is moved through the namenode  2) Maintaining overall health: - Periodic health checks of datanodes. - Block re-replication & rebalancing. - Garbage collection  If namenode data is lost, all files on filesystem cannot be retrieved since there is no way to reconstruct them from raw block data. However, Hadoop does provide backups and secondary namenodes (out of syllabus) | | | | | |
| Namenode + Master Node & Quiz | | | | | | A diagram of a computer system  Description automatically generated | | Partitioner determines which keys are processed on which reducers only.  Shuffle stage is not always run within a single node, but runs in the worker nodes |
| Ass1 | 1. State 2 optimizations which decrease the amount of data (e.g., in bytes) exchanged between the mappers and reducers for this program.  - Use in-mapper combiners, locally aggregating the tuples in a hash data structure in the mapper before emitting them (or alternatively, locally aggregating the tuples per line);  - Replace the strings 'Singapore' and 'United States' with shorter strings like 'S' and 'U';  - Compression of map outputs can be used to reduce the size of map outputs, and thus the data exchanged between mappers and reducers.  - Filter away the stopwords in the mapper rather than in the reducer.  Common errors:  - Filtering the data in the reducers (does not reduce data exchanged)  - Applying combiners in the reducer function (incorrect / does not reduce data exchanged)  - Filtering tuples by the > 500 threshold in the mapper (leads to incorrect output, as a tuple may appear 250 times in mapper 1 and 250 times in mapper 2, which should not be filtered) | | | | | | 2. By default, Hadoop uses 1 reducer (that is, 1 reduce task). In fact, you can increase the number of reducers: how does it change the final outputs in the `/content/output/` directory? Explain why this occurs, by stating what data is received as input for each of the reducers.  There are now two output files, each with a separate set of keywords. This is because the keys output from the mappers are partitioned into two parts (by the partitioner), which are handled by two reducers. Each reducer receives a list of lines corresponding to all key value pairs for the set of keys it is responsible for.  Common mistakes here include:  - Saying that the occurrences of a specific keyword can be split up over the 2 reducers, thus making it appear less than 500 times. (Note: First, consider the case where we use the keyword as the mapper's emitted key. Then, since the partitioner partitions each key to only one of the reducers, all occurrences of each keyword (e.g., 'Hi') will go to the same reducer. Thus, it is not possible for the occurrences of a keyword to be split over 2 reducers. However, there are some less common cases (e.g., if you use a composite key of keyword and country) where this can happen, as the composite keys of ('Hi', 'Singapore') and ('Hi', 'United States') can be assigned to different reducers). | |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Relational DB and MapReduce | | | | | Relational DB comprised of tables. Each table = a relation = collection of tuples (rows). Each tuple consists of multiple fields  Star Schema - Fact and dimension tables. Join on PK. Fact are things that changes. Dim should not be changing like Country Code  Projection in MapReduce: take in tuple (with tuple ID as key) and emit new tuples with appropriate attributes, . No reducer needed (no need shuffle step)  Selection: SELECT \* FROM table WHERE (predicate),  Selection in MapReduce: take in tuple (w tuple ID as key) and emit only tuples that meet the predicate. No reducer needed | | |
| Group by | | SELECT id, AVG(price) FROM table GROUP BY id  MapReduce: Map over tuples, emit <id, price> . Framework auto groups tuples by key. Compute average in reducer. Optimize w combiners | | | | | |
| Joins | Many-to-Many r/s, One-to-Many, One-to-One | | | | | | |
| A diagram of a data processing process  Description automatically generatedMtd 1) Broadcast (or 'Map'') Join: Requires 1 of the tables to fit in memory  - all mappers store a copy of the small table (for efficiency: convert it to a hash table, w keys as the keys we want to join by)  - They iterate over the big table, and join the records w the small table  Mtd 2) Reduce-side (or 'Common') Join: Don't require a dataset to fit in memory, but slower than broadcast join  - Diff mappers operate on each table, & emit records, w key as the variable to join by  A diagram of a task  Description automatically generatedA diagram of a function  Description automatically generated with medium confidenceIn reducer: use secondary sort to ensure that all keys from table X arrive before table Y. Then hold keys from table X in memory and cross them w records from table Y | | | | | | |
| Similarity Search | | | Distance measure: measure dist btw obj x and y, use function d(x, y). Similarity measures are opp: lower dist = higher similarity, vice versa  Euclidean dist = d(**a, b**) = . Manhattan dist = d(**a, b**) =  Cosine similarity = s(**a**, **b**) = . Only considers dirn: cosine similarity don't change if we scale a or b  Jaccard Similarity (btw sets A and B) = . Jaccard Distance = | | | | |
| Finding similar documents | | | | All pairs similarity: Given a large num N of documents, find all "near duplicate" pairs, e.g. w Jaccard dist below a threshold  Similarity search: OR: given a query document D, find all documents which are "near duplicates" with D  Applications: Mirror websites, or approximate mirrors - Don't want to show both in search results. Or cluster news articles by topic | | | |
| A grid of numbers and letters  Description automatically generated1. Shingling: Convert documents to sets of short phrases  - k-shingle/k-gram for a document is a seq of k tokens that appears in the doc  - E.g. k = 2, document D1 = "cat is happy". Set of 2-shingles: C1 = S(D1) = {"cat is", "is happy"}  - Often represented as a matrix, where columns = documents, rows = shingles  - Measure similarity btw documents as Jaccard similarity = sim(D1, D2) =  2. Min-Hashing: Convert these sets to short "signatures" of each document, while preserving similarity  - "Signature" = block of data representing the contents of a document in a compressed way  - Documents w same signature are candidate pairs for finding near duplicates  - Motivation for MinHash: to compute pairwise Jaccard similarities for every pair of docs = N(N-1)/2 which is slow  - MinHash gives fast approx to result of using Jaccard similarities to compare all pairs of docs | | | |
| MinHash: hash each column C to a small signature h(C) s.t.:  1) h(C) is small enough that the signature fits in RAM 2) highly similar documents usually have same signature  So goal is to find hash function h, s.t. if sim(C1, C2) is high, then h(C1) = h(C2) w high prob, if sim(C1, C2) is low, then h(C1) ≠ h(C2) w high prob  So hash function h maps each shingle to an integer. Then compute the min of these integers  - Pr[h(C1) = h(C2)] = Jaccard-Sim(C1, C2). (intuition: each of the shingles has same prob of having smallest hash value, so Pr[h(C1) = h(C2)] = intersection of shingles/union = Jaccard similarity) | | | |
| Candidate pairs = docs w same final signatures. Can directly output them OR compare 1 by 1 to check if they are actually similar pairs  In practice, usually use multiple has functions (e.g. N = 100) and generate N signatures for each docs. "candidate pairs" = those having "sufficient" (e.g. ≥ 50) matching signatures | | | |
| MapReduce Implemen-tation | | | | Map: 1) Read over doc and extract its shingles. 2) Hash each shingle and min of them = MinHash. 3) Emit <signature, document\_id>  Reduce: 1) Receive all documents w a given MinHash signature. 2) Generate all candidate pairs from these documents.  3) (Optional) Compare each such pair to check if they are actually similar) | | | |
| Clustering | | | | | Separates unlabelled data into groups of similar points. Clusters should have high intra-cluster similarity and low inter-cluster similarity | | |
| K-means algo | | 1. Initialization: Pick K random points as centers  2. Repeat: { a) Assignment: assign each point to nearest cluster  b) Update: move each cluster center to average of its assigned points }  Stop if no assignments change | | | | | n = num of points, m = num of iterations,  d = dimensionality, K = num of clusters |
| A black and white text with red text  Description automatically generatedThis MapReduce job performs a Single iteration of k-means.  It receives current positions of cluster centers as inputs.  Disk I/O exchanged btw mappers and reducers = O(nmd) | | | | A black and white text on a white background  Description automatically generated With in-mapper combiner  Disk I/O exchanged btw mappers and reducers = O(Kmd) | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| NoSQL | NoSQL = non-relational database. SQL here refers to relational databases, not the querying language  NoSQL systems can use SQL-like querying language  1) Horizontally scalablility. 2) Replicate/distribute data over many servers (can partition data into diff machines). 3) Simple call interface.  4) Often weaker concurrency model than RDBMS 5) Efficient use of distributed indexes and RAM. 6) Flexible schemas  1) and 2) to deal w volume and velocity of data. 6) to deal w variety   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | Types of NoSQL DB | Key-value stores | Document stores | Wide column DB | Graph DB | Vector DB | | e.g. | redis, DynamoDB | mongoDB, CouchDB | Cassandra, HBase | Neo4j, HyperGraphDB | Milvus, redis | | | | | | | |
| Key-value Stores | | Keys are usually primitives and can be queried (e.g. ints, strings, raw bytes,...)  Values can be primitive or complex; usually cannot be queried (e.g. ints, strings, lists, JSON, HTML fragments, BLOB (basic large object), ...) | | | | |  |
| API: 1) Get – fetch value associated w key. 2) Put – set value associated w key  Optional operations: Multi-get, Multi-put, Range queries | | | | |
| Suitable for: - Small continuous read and writes (fast read and write)  - Storing 'basic' info (e.g. raw chunks of bytes), or no clear schema  - When complex queries are not/rarely required | | | | |
| Applications: Storing user sessions / Caches / User data that is often processed individually | | | | |
| Implementation:  1) Non-persistent: Just a big in-memory hash table (in memory faster than disk read or write)  - Memcached, Redis (although can back up data to disk periodically, not the main purpose)  2) Persistent: Data stored persistently to disk  - RocksDB, Dynamo, Riak | | | | |
| Document Stores | | DB can have multiple collection, which can have multiple documents   |  |  |  |  | | --- | --- | --- | --- | | MongoDB | Collections | Documents (BSON; binary JSON) | Fields | | SQL | Tables | Rows | Columns | | | | A computer screen with text and numbers  Description automatically generated  MongoDB syntax  $: signify special keyword, gt: greater than  If set to 0 = get all other fields other than the one set to 0  Specify update criteria, Specify update action  Set update option  Delete criteria | | |
| Different documents can have diff fields  Can be nested, i.e. JSON objects as values | | |
| Unlike (basic) key value stores, document stores allow some querying based on content of document  - CRUD = Create, Read, Update, Delete | | |
| Create: db.users.insert ( { name: "sue", age: 25, status: "A" } )  Read: db.users.find( { age: { $gt: 18 } },  { name: 1, address: 1 } ).limit(5)  Update: db.users.update ( { age: { $gt: 18 }, { $set: { status: "A" },  { multi: true } } } )  Delete: db.users.remove ( { status: "D" } ) | | |
| Wide Column Stores | | A screenshot of a computer  Description automatically generatedRows describe entities  Related groups of columns are grouped as col families  Sparsity: If col not used for a row, don't take up space (unlike RDS, need space to store NULL)  Small table on right: historical values; at time t1: col2 value is ...; t2: col2 value is ... | | | | | |
| Graph DB | | Nodes and Edges | | | | | |
| Vector DB | | Stores vectors (each row represent a point in d dimensions): usually dense, numerical and high dimensional  Allow fast similarity search, i.e. given query, retrieve similar neighbors from DB: Uses locality-sensitive hashing (LSH; related to min hashing)  Features: Scalability, real-time updates, replication  AI/ML uses this to store converted text (embeddings) in form of vectors; useful for search, recommendation, clustering  - Also can use to store images converted into embeddings | | | | | |
| Eventual Consistency | | | Strong Consistency: any reads immediately after an update must give same result to all observers  Eventual consistency: if system is functioning and we wait long enough, eventually all reads will return the last written value  Strong consistency make use of locks to prevent reads until change is propagated, i.e. availability is lower | | | | |
| RDBMS provide stronger (ACID = atomicity, consistency, isolation, durability) guarantees # consistency here refers to constraint  Many NoSQL system relax this to weaker BASE (Basically Available, Soft state, Eventually consistent)  Basically Available: basic reading and writing ops are available most of the time  Soft state: without guarantees, we only have some probability of knowing the state at any time (not sure when changes are propagated)  Although NoSQL systems allow for weaker consistency guarantees, recent systems are often configurable, i.e. can be configured for multiple diff consistency levels (including strong) – 'tunable consistency' | | | | |
| Duplication | | | RDBMS will use joins to join 2 tables. Only some recent NoSQL DBs support joins  To achieve 'joins': use duplication/denormalization  - Storage is cheap, so just store more data in '1 huge table' to improve efficiency  - Design that table around queries we expect to receive - Problems come when we need to propagate changes/updates to multiple tables | | | | |
| Pros and Cons | | Pros: - Flexible and dynamic schema - Horizontal scalability  - High performance and availability (due to eventual consistency and fast reads/writes) | | Cons: No declarative query language (e.g. joins): have to be handled on application side, add additional programming  - Weaker consistency guarantees: may receive stale data | | | |
| |  |  |  | | --- | --- | --- | | Pros | Performance | Sacrifice strong consistency while aiming for low latency and high availability (due to removing need for locks and other concurrency mechanisms) | | Scalability | Horizontally scalable (due to sacrificing strong consistency) RDBMS are vertically scalable (need more powerful machines to make it faster) | | Cons | Outdated data | Outdated data may cause mistakes/require additional work to handle on the app side | | | | | | |
| Key-Value store:  - Improves scalability and efficiency – writing or reading user pages is faster  - No need for complex queries or based on the contents of user pages – just reads & writes  - May be acceptable for user pages to be slightly stale – then eventual consistency is acceptable | | | | | | Document store:  - Flexible schema may be beneficial (e.g. special types of vehicles may require diff sets of fields)  - Unlike key-value stores, document stores are more suitable for queries based on fields of a document | |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| NoSQL systems as distributed databases | Scalability: allow DB sizes to scale simply by adding more nodes  Availability/Fault Tolerance: If 1 node fails, others can still serve requests  Latency: Usually, each request is served by the closest replica, reducing latency, when DB is distributed over a wide geographical area  - Data Transparency: Users don't need to know how data is physically distributed, partitioned or replicated  - Data Transparency: Query that works on a single node DB should still work on a distributed DB  Assumptions: All nodes in a distributed DB are well-behaved, i.e. follow protocol that is designed for them; not 'adversarial' or trying to corrupt the DB (OOS, if can't trust the other nodes, we need a 'Byzantine Fault Tolerant' protocol, e.g. Blockchains w decentralized ledger) | | | | | | | |
| Distributed DB Architectures | | | A black and white image of a computer  Description automatically generatedCPU  Memory  Disk  E.g. Single node DBMS | A computer network diagram with a red sign and text  Description automatically generated with medium confidenceSupercomputers | | A screen shot of a computer  Description automatically generatedCloud databases – Oracle RAC, snowflake | | NoSQL (mostly) – Cassandra, Redis, mongoDB, CouchDB, MySQL |
| Data Partitioning | | Table Partitioning: Put diff tables/collections on diff machines. Problems: scalability – each table cannot be split across multiple machines  Horizontal Partitioning/Sharding: Diff tuples stored in diff nodes  Partition/shard key: variable to decide which node each tuple is stored on: tuples w same shard key is on the same node  Chosing partition key: If we often need to filter tuples or "group by" based on a column, then use that col as partition key  E.g. if we filter tuples by id=100, and id is partition key, then all id=100 is on same partition. Data from other partitions can be ignored (i.e. partition pruning), saves time as we don't have to scan tuples in other partitions  If col has very little unique values/*low cardinality*, don't use as partition key (as won't be able to split data into many diff partitions/*lack of scalability*). Also if some values have *too high frequency*, then don't use as partition key. (Can be mitigated using composite keys) | | | | | | |
| Horizontal Partitioning | | 1) Range Partition: split partition key based on range of values (e.g. Partition 1: 1 ≤ id ≤ 100. Partition 2: 101 ≤ id < 200)  Good if we need range-based queries (i.e. user queries for id < 50, all data in partition 2 can be ignored/partition pruning)  Bad: can lead to imbalanced shards (e.g. if many rows have id = 0)  Splitting the range is auto handled by a balancer (tries to keep shards balanced) | | | | | | |
| 2) Hash Partition: hash partition key, then divide into partitions based on range (e.g. Partition 1: 1 ≤ hash(id) ≤ 16. Partition 2: 17 ≤ hash(id) < 32)  Hash fn auto spreads out partition key values roughly evenly  However, if we add/remove a node/partition, have to completely redo partition, i.e. a lot of data have to be moved, which is inefficient | | | | | | |
| A diagram of a network  Description automatically generated2a) Consistent Hashing solve this problem: Think of output of hash functions as lying on a circle  Each node has a 'marker'/rectangles  Each tuple is placed on the circle, and assigned to the node that comes clockwise-after it  a) Delete a node: re-assign all its tuples to the node clock-wise after this one  b) Add a node: add a new marker, and re-assign all tuples which now belong to the new node  *Simple replication strategy:* replicate a tuple in the next few additional nodes (e.g. 2) clockwise after the primary node used to store it  *Multiple markers:* Can have multiple markers per node. For each tuple, still assign it to the marker nearest to it in the clockwise dirn  Benefit of multiple markers: if we remove a node, its tuples will not all be reassigned to the same node. Hence balancing load better | | | | | | |
| Architecture of MongoDB | | A diagram of a computer  Description automatically generatedE.g. of read/write query  1. Query issued to a router (mongos) instance  2. With help of config server, mongos determines which shards to query  3. Query sent to relevant shards (partition pruning)  A diagram of a computer  Description automatically generated- If query not based on partition/shard key, i.e. relevant to all shards, then query will go to all shards  4. Shards run query on their data, and send results back to mongos  5. mongos merge query results and returns merged results back to app  Replication in MongoDB: Common config (1 primary, 2 secondary)  a) Writes: "Primary" receives all write operations  - Records writes onto its "operation log"  - Secondaries then replicate this operation log, apply it to their local copies of data, then acknowledge the operation  b) Reads: User configure read preference (read from secondaries – default, or from primary)  - Reading from secondaries decr latency and distribute load (improving throughput of entire system), but allows for reading of stale data (only has eventual consistency)  c) Elections: If primary node fails, the nodes "conduct an election", a protocol to choose 1 of the secondaries to be promoted to primary | | | | | | |
| Reasons for Scalability & Performance of NoSQL | | | Horizontal partitioning: more data can be partitioned into more and more shards (even if individual tables becomes very large)  - Horizontal partitioning improves speed due to parallelization  Duplication/denormalization: NoSQL queries go to only 1 table, unlike RDBMS where queries might look up multiple tables (joins)  Relaxed consistency guarantees: prioritize availability over consistency – can return slightly stale data | | | | | |
| Range partitioning | | | Benefits:  - If we need to run filter or group by queries based on num of products bought (e.g. frequent buyers vs new buyers), this partitioning scheme is efficinet (as info about buyers w similar num of products bought is co-located, allowing faster queries w/o need to scan entire database. | | Disadvantages: - May lead to highly imbalanced partitions, e.g. if a large num of users have only bought 1 product  - Maintenance: when customer starts buying a lot of products, they need to be moved from 1 partition to another, leading to additional overhead  - Other queries: other user characteristics may be more impt, like age, geography, ... Queries grouping by such characteristics would need to scan multiple partitions | | | |
|  | | | Document Store: could be useful for storing course info or student info. Document stores like MongoDB provide flexible schema to accomodate diff course characteristics (e.g. some courses may have prerequisites/cross-listings, others may not). Similarly, some students may have input various profile info, while others may not. | | | | Graph DB: could be useful if certain use-cases crucially involve relationships: e.g. recommending courses to students, recommending sutdy groups for students, managing complex prerequisites / time conflicts btw courses to recommend a timetable, ... | |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Hadoop vs Spark | A circular object with three squares  Description automatically generatedIssues w Hadoop Mapreduce: (MR on diag = MapReduce)  - Network and disk I/O costs: intermediate data has to be written to local disk and shuffled across machines, which is slow  - Not suitable for iterative (i.e. modifying small amounts of data repeatedly) processing, as each individual step has to be modelled as a MapReduce job  Instead, Spark stores most of its intermediate results in memory, making it faster, especially for iterative processing  - When memory is insufficient, Spark spills to disk which requires disk I/O | | | | | |
| Ease of programmability: Hadoop have to define mapper, reducer, main function. | | Spark: *val file = sc.textFile("hdfs://..")*  *val counts = file.flatMap(line => line.split(" ")).map(word => (word, 1)).reduceByKey(\_+\_)*  *counts.save("...")* | | | |
| Spark Components and API Stack | | |  |  |  |  | | --- | --- | --- | --- | | Spark SQL and DataFrames + Datasets | Spark Streaming (Structured Streaming) | Machine Learning  MLlib | Graph Processing  Graph X | | Spark Core and Spark SQL Engine: Scale, SQL, Python, Java, R | | | | | | | | |
| Spark Architecture & Evolution of Spark APIs | A diagram of a computer process  Description automatically generatedDriver Process responds to user input, manages Spark application etc., and distributes work to Executors, which run the code assigned to them and send the results back to the driver  Cluster Manager (can be Spark's standalone cluster manager, YARN, Mesos or Kubernetes) allocates resources when the application requests it  In local mode, all these processes run on the same machine  Spark APIs:   |  |  |  | | --- | --- | --- | | Resilient Distributed Datasets (RDDs) (2011) | DataFrame (2013) | DataSet (2013) | | Collection of JVM objects  Functional operators (map, filter, etc) | Collection of Row objects  Expression-based operations  Logical plans and optimizer | Internally rows, externally JVM objects  Almost the "Best of both worlds": type safe + fast | | | | | | |
| RDDs | 1) Resilient: achieve fault tolerance through lineages. 2) Distributed Datasets: collection of objects distributed over machines  Due to 2), transformations and actions are executed in parallel. Results only sent to the driver in the final step | | | | | |
| 1) RDDs are immutable, i.e. cannot be changed once created  *dataRDD = sc.parallelize(['Alice', 'Bob', 'Carol', 'Daniel'], 3)* # Create an RDD of names, distributed over 3 partitions.  OR 1) Read in RDDs. *sc.textFile("File.txt")* reads file on each worker node in parallel, not on the driver node  2) Transformations are a way of transforming RDDS into RDDs. Transformations are lazy, i.e. not executed yet, until an action is called on it  With lazy transformations, Spark can optimize query plan to improve speed (e.g. removing unneeded operations)  E.g. of transformations: map, order, groupBy, filter, join, select, cache. *nameLen = dataRDD.map(lambda s: len(s))*  3) Actions trigger Spark to compute a result from a series of transformations. Action ask Spark to retrieve all elements of RDD to the driver node. E.g. of actions: show, count, save, collect. *nameLen.collect()* | | | | | |
| Caching | *lines = sc.textFile("hdfs://..")*  *errors = lines.filter(lambda s: s.startswith("ERROR"))*  *messages = errors.map(lambda s: s.split("\t")[2])*  *messages.cache()*  *messages.filter(lambda s: "mysql" in s).count()*  *messages.filter(lambda s: "php" in s).count()* | | | Read and partition data into diff blocks  Driver distributes tasks to workers  Workers read from their HDFS block  Worker process & Cache data (transformation)  Workers sent results back to driver  Driver now sends another task to workers, workers can now process from Cache & send results back. No need to recompute | | |
| *cache()*: saves an RDD to memory (of each worker node)  *persist(options)*: can be used to save an RDD to memory, disk, or off-heap memory  Should cache RDD if it is expensive to compute and needs to be reused multiple times  If worker nodes have not enough memory, they will evict the "least recently used" RDDs | | | | | |
| Directed Acyclic Graph (DAG) | A diagram of a map  Description automatically generatedInternally, Spark creates a graph to represents all RDD objects and how they will be transformed  Transformation construct this graph; actions trigger computations on it  A comparison of a file type  Description automatically generated with medium confidenceNarrow dependencies: each partition of the parent RDD is used by at most 1 partition of the child RDD. E.g. map, flatmap, filter, contains  Wide dependencies: each partition of the parent RDD is used by multiple partitions of the child RDD. E.g. reduceByKey, groupBy, orderBy  In DAG, consecutive narrow dependencies are grouped tgt as "stages"  Within stages, Sparks performs consecutive transformations on the same machines  Across stages, data needs to be shuffled, i.e. exchanges across partitions (similar to map-reduce), and hence involves writing intermediate results to disk  Minimizing shuffling can improve performance | | | | | |
| Lineage and Fault Tolerance | Unlike Hadoop, Spark does not use replication to allow fault tolerance.  Spark tries to store all the data in memory, not disk. Memory capacity is much more limited than disk, so duplicating all data is expensive.  Lineage approach: if a worker node goes down, replace it by a new worker node and use DAG to recompute data in lost partition | | | | | |
| DataFrames | Table of data, similar to tables in SQL or DataFrames in pandas  Compared to RDDs, this is a higher level interface, e.g. it has transformations that resemble SQL operations  - DataFrames (and Datasets) are recommended interface for working w Spark – easier to use than RDDs, and almost all tasks can be done w them, while only rarely using the RDD functions  - However, all DataFrame operations are still ultimately compiled down to RDD operations by Spark | | | | | |
| *2015 = spark.read.option("inferSchema", "true")\*  *.option("header", "true").csv("mnt/defg/flight-data/csv/2015.csv")*  *2015.sort("count").take(3)*  *2015.createOrReplaceTempView("data\_2015")*  *maxSql = spark.sql("""*  *SELECT Country, sum(count) AS total FROM data\_2015*  *GROUP BY Country ORDER BY sum(count) DESC LIMIT 5*  *""")*  *maxSql.collect()* | | | | | # sort by count and output first 3 rows  Using SQL queries. Takes in DataFrame and returns DataFrame  OR *from pyspark.sql.functions import desc*  *2015.groupBy("Country".sum("count")\*  *.withColumnRenamed("sum(count)", "total").sort(desc("total")).limit(5).collect()* |
| Datasets | Datasets similar to DataFrames, but are type-safe  - In Spark (Scala), DataFrame is just an alias for Dataset[Row]  - Datasets not available in Python and R, as they are dynamically typed languages  # Dataset *flights* is type safe – type = "Flight"  # return objects of "Flight" class, instead of Row objects | | | | *case class Flight(DEST\_COUNTRY\_NAME: String, ORIGIN\_COUNTRY\_NAME: String, count: BigInt)*  *val flightsDF = spark.read.parquet("/mnt/defg/flight-data/parquet/2010- summary.parquet/")*  *val flights = flightsDF.as[Flight]*  *flights.collect()* | |

|  |  |  |  |
| --- | --- | --- | --- |
| Spark | | A diagram of a software development process  Description automatically generatedDesign Philosophy: Speed, Ease of use, Modularity, Extensibility  Spark SQL: Unifies Spark components and permits abstraction to DataFrames/Datasets in Java, Scala, Python and R  Keep track of schema and support optimized relational operations  RDD: - Instruct Spark how to compute the query  - Intention is opaque to Spark - Spark don't understand structure of data in RDDs  DataFrame: - Tell Spark what to do, instead of How to do  - Code is more expressive as well as simpler (uses a domain specific language, DSL, similar to python pandas; Use high-level DSL operators to compose the query)  - Spark can inspect or parse this query, understand our intentions, then optimize or arrange operations for efficient execution | |
| Catalyst Optimizer | | Diagram of a diagram  Description automatically generatedTakes a computational query and converts it into an execution plan through 4 transformational phases: 1. Analysis, 2. Logical optimization, 3. Physical planning, 4. Code generation  *usersDF = ...; eventsDF = ...*  *joinedDF = users.join(events, users("id") === events("uid"))*  *.filter(events("date") > '2015-01-01")*  Logical Plan -> Physical Plan  *A diagram of a computer flowchart  Description automatically generated*A diagram of a diagram  Description automatically generated  Physical Plan w Predicate Pushdown & Col Pruning  *A diagram of a business  Description automatically generated* | |
| Spark ML | Data Quality  Missing Values: Info not collected/Missing at random. If data is missing not at random, then missingness may itself be impt info  To handle missing values: 1) Eliminate objects/rows w missing values. 2a) Fill based on mean/median of attr  2b) Fill by fitting regression model to predict attr given other attr. 2c) Dummy variables: insert new col w 1 if attr missing, 0 otherwise  *from pyspark.ml.feature import Imputer*  *imputer = Imputer(inputCols=['a', 'b'], outputCols=['out\_a', 'out\_b']). model = imputer.fit(df) model.transform(df).show()* | | |
| Data Preprocessing/Feature Engineering  1) Categorical Encoding  Convert categorical feature to numerical features. Have some ordinal r/s or inherent order among categories  E.g. [Low, Medium, High] converted to [0, 1, 2] | | 2) One Hot Encoding  Convert discrete feature to a series of binary features  When there's no ordinal r/s among categories  [Grp2, Grp1, Grp3] -> [Grp1: [0, 1, 0], Grp2: [1, 0, 0], Grp3: [0, 0, 1]] |
| Data Preprocessing: Normalization  1) Clipping: [1,2,3,1000] -> [1,2,3,3]  2) Log transform: log(1+x) | | 3) Standard Scale:  4) Max Min Normalization: |
| Logistic Regression: Sigmoid Fn = . w = weights, b = bias  Prediction (probability) =  On training data, fit parameters of model (w & b) by minimizing a loss/cost fn J(w, b)  Cost fn, J(w, b) = Cross Entropy Loss. The closer the model prediction/probability to the label, the lower the loss value  Minimize cost fn J using gradient descent. Follow steepest downward slope until convergence/improvement below a fixed threshold  Given n training samples w d features: Update: , is step size  Weights Vector update: . Can be made into parallel gradient descent (use map-reduce) | | |
| Evaluation: Confusion matrix: columns = actual labels; rows = predicted labels  Accuracy = . Sensitivity = (fraction of actual positive detected). Specificity = (fraction of actual negatives detected) | | |
| Pipelines. For better code reuse. Easier to perform CV and hyperparameter tuning  Building Blocks: Transformers (map df to df): One-hot encoding, tokenization. Transformer obj has transform() mtd, to perform transformation  Transformers typically output a new df which append their result to original df.  So fitted model is also a "Transformer" that transforms a df into 1 with predictions appended  Building Blocks: Estimator (algo which takes in data, and outputs a fitted model): Have a fit() mtd, to return a Transformer  *from pyspark.ml.classification import LogisticRegression*  *training = spark.read.format("libsvm").load("data/mllib/sample\_libsvm\_data.txt"); lr = LogisticRegression(maxIter=10)*  *lrModel = lr.fit(training); str(lrModel.coefficients); str(lrModel.intercept)* | | |
| Pipeline chains tgt multiple Transformers and Estimators to form an ML workflow. Pipeline is an Estimator. When Pipeline.fit() is called:  Starting from beginning of pipeline: { For Transformers, call transform(); For Estimators, call fit() }.  Output = estimated pipeline model (of type PipelineModel)  *tokenizer = Tokenizer(inputCol="col1", outputCol="col2"); hashingTF = HashingTF(inputCol=tokenizer.getOutputCol(), outputCol="col3")*  *lr = LogisticRegression(maxIter = 10, regParam=0.001); pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])*  *model = pipeline.fit(training); pred\_train = model.transform(training); pred\_train.drop('rawPrediction').show(truncate=False)*  *pred\_test = model.transform(test); predictionAndLabels = pred\_test.select("prediction", "Y")*  *evaluator = MulticlassClassificaionEvaluator(metricName = "accuracy"); print(str(evaluator.evaluate(predictionAndLabels))* | | |
| Regression Models  (Eg in practical 2) | | Linear Regression, Decision Tree Regressor, Random Forest Regressor  Evaluation: MAE = . MSE = . RMSE =  R2 (closer to 1 = better fit) = . . . | |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Streaming | | | Process input as it is received. Compared to offline/batch approaches that operate on full dataset, all at once  Input elements enter at a rapid rate from input ports (e.g. sensor, TCP connection, from file stream, message queue)  Elements of stream sometimes referred to as 'tuples'. Stream is potentially infinite; i.e. system cannot store the entire stream  Data is high volume and constantly arriving over time | | | | | | |
| Stateful Stream Processing: - not just perform trivial record-at-a-time transformations  - has ability to store and access intermediate data  - state can be stored and accessed in many diff places (program variables, local files, embedded/external databases) | | | | | | |
| Spark | Micro-Batch Stream Processing: - divide data from input stream into micro batches  - each batch is process in Spark cluster in a distributed manner  - small deterministic tasks generate output in micro-batches | | | | | | Advantages: - quickly and efficiently recover from failures  - deterministic nature ensures end-to-end exactly-once processing guarantees  Disadvantages: - latencies of a few seconds  - App may incur > few seconds delay in other parts of pipeline | | |
| A diagram of a process flow  Description automatically generatedStructured Streaming programming model: data stream as an unbounded table (new records appended to unbounded table)  A diagram of a process flow  Description automatically generated- Users express query on streaming data using a batch-like API and Structured Streaming incrementalizes them to run on streams  To define Streaming Query:  1) Define input sources  2) Transform data  3) Define output sink and output mode  - Output writing details (where and how to write output)  - Processing details (how to process data and how to recover from failures)  4) Specify processing details  - Triggering details: when to trigger discovery and processing of newly available streaming data  - Checkpoint location: store streaming query process info for failure recovery  5) Start query . Similarly, user query converted to logical plan -> optimized plan -> series of incremental executions plans | | | | | | | | |
| Stateless transformations: process each row individually w/o needing any info from previous rows  Projection ops: select(), explode(), map(), flatMap(). Selection ops: filter(), where()  Stateful transformation: e.g. df.groupBy().count()  In every micro-batch, incremental plan adds count of new records to previous counts generated by previous micro-batch  Partical count communicated btw plans is the state  State is maintained in memory of Spark executors and is checkpointed to the configured location to tolerate failures | | | | | | | | |
| Event time = time event actually happened. Processing time: time event is received at processing machine  - Event time completely decouples the processing speed from the results  - Ops based on event time are predictable and their results are deterministic  - Event time window computation will yield the same result no matter how fast stream is processed on when the events arrive at operator  - Qn now is how long to wait before we can be certain that we have all events that happened before a certain point in time: Watermarks | | | | | | | | |
| Stateful Streaming Aggregations  1) Aggregations not based on time  a) Global aggregations: *runningCount = sensorReadings.groupBy().count()*  b) Grouped aggregations: *baselineValues = sensorReadings.groupBy('sensorId').mean('value')*  c) All built-in aggregation fns in DataFrames are supported (*sum, mean, stddev, countDistinct, collect\_set, approx\_count\_distinct, ...)*  2) Aggregations w Event-Time Windows | | | | | | | | |
| Mapping of event time to 5 min tumbling windows  *A diagram of a timeline  Description automatically generatedsensorReadings.groupBy('sensorId', window('eventTime', '5 minute')).count()* | | | | | | | Mapping of event time to overlapping windows of length 10 mins and sliding interval 5 mins  *sensorReadings.groupBy('sensorId', window('eventTime', '10 minute', '5 minute')).count()*  A diagram of a timeline  Description automatically generated | |
| *sensorReadings \*  *.withWatermark('eventTime', '10 minutes') \*  *.groupBy('sensorId', window('eventTime', '10 minutes', '5 minutes')) \*  *.count()*  Trigger computation after every 5 minutes  Computation is for past 10 minutes  Watermark = max event time before this time frame for which computation was triggered - 10 mins  Data in time frame before watermark -> reject and don't include in computation  Data in time frame after/including watermark -> include in computation | | | | | | | | |
| Performance tuning | | | | | Besides tuning Spark SQL engine, other things to consider:  Cluster resouce provisioning appropriately to run 24/7. Num of partitions for shuffles to be set much lower than batch queries  Setting source rate limits for stability. Multiple streaming queries in same Spark application | | | | |
| Flink | | | | | A diagram of a network  Description automatically generated with medium confidenceInput must be events logs or message queue (kafka)  Dataflow model: Data exchange strategies  System architecture: Flink is a distributed system for stateful parallel data stream processing  ResourceManager = Cluster Manager  Job manager = driver  A diagram of a network  Description automatically generated with medium confidenceTask manager = worker node  TaskManager can execute several tasks at the same time:  - tasks of same operator (data parallelism)  - tasks of diff operators (task parallelism)  A diagram of a task manager  Description automatically generatedA diagram of a job manager  Description automatically generated- tasks from a diff app (job parallelism) | | | | |
| TaskManager offers a certain num of processing slots to control the num of tasks it is able to concurrently execute  A processing slot can execute 1 slice of an app - 1 parallel task of each operator of the app | | | | |
| Data transfer in Flink | | | | | Tasks of a running application are continuously exchanging data  TaskManagers take care of shiping data from sending tasks to receiving tasks  Network component of a TaskManager collects records in buffers before they are shipped  Microbatch in spark in synchronous (blocking) vs Flink: each TaskManager maintain their own buffer asynchronously. Once buffer is full, send data (to minimize network traffic) | | | | |
| Event-Time Processing | | | | | A diagram of a record time stamp  Description automatically generatedTimestamps: Every record must have an event timestamp  Watermarks: used to derive the current event time at each task in an event-time app  In Flink: watermarks are implemented as special records holding a timestamp as a Long value. Watermarks flow in a stream of regular records w annotated timestamps  *earlyFiring (watermark says all events has appeared, but actually not), lateFiring* | | | | |
| State Management | | | | | 1) Stateful stream processing task: all data maintained by a task and used to compute results of a function belong to the state of the task  2) Operator state: scoped to an operator task. All records processed by the same parallel task have access to the same state.  A diagram of a computer process  Description automatically generated- Operator state cannot be access by another task of the same or a diff operator  3) Keyed State: maintains 1 state instance per key value  - partitions all records w the same key to the operator task that maintains the state for this key  A diagram of a task  Description automatically generated | | | | |
| State Backend  a) Local State Management: - a task of a stateful operator typically reads and updates its state for each incoming record  - each parallel task locally maintains its state in memory to ensure fast state accesses  b) Checkpointing: - TaskManager process may fail at any point, hence its storage must be considered volatile  - Checkpointing the state of a task to a remote and persistent storage  - remote storage for checkpointing could be a distributed filesystem or a database | | | | |
| Checkpoints | | | | Consistent checkpoints: similar to Spark micro-batch checkpoints. "Stop-the-world"  1. Pause ingestion of all input streams  2. Wait for all in-flight data to be completely processed, i.e. all tasks have processed all their input data  3. Take a checkpoint by copying state of each task to a persistent storage. Checkpoint is complete when all tasks have finished their copies  4. Resume ingestion of all streams | | | | | |
| Failure recovery from a consistent checkpoint  1. Restart whole application. 2. Reset states of all stateful tasks to the latest checkpoint. 3. Resume processing of all tasks | | | | | |
| Flink’s Checkpointing Algorithm: based on the Chandy–Lamport algorithm for distributed snapshots  - does not pause the complete application but decouples checkpointing from processing  - some tasks continue processing while others persist their state  - uses a special type of record called a *checkpoint barrier*  - checkpoint barriers are injected by source operators into the regular stream of records and can't overtake or be passed by other records  - A checkpoint barrier carries a checkpoint ID to identify the checkpoint it belongs to and logically splits a stream into two parts  - All state modifications due to records that precede a barrier are included in the barrier’s checkpoint and all modifications due to records that follow the barrier are included in a later checkpoint. | | | | | |
| Streaming app w 2 stateful sources, 2 stateful tasks, 2 stateless sinks  A diagram of a number system  Description automatically generated | | | | | A diagram of a algorithm  Description automatically generatedJobManager initiates a checkpoint by sending a message to all sources |
| Sources checkpoint their state & emit a checkpoint barrier  A diagram of a work flow  Description automatically generated | | | | | Tasks wait to receive a barrier on each input partition  Records from input stream from which a barrier already arrived are buffered. All other records processed normally  A diagram of a process  Description automatically generated |
| Tasks checkpoint their state once all barriers have been received, then forward the checkpoint barrier  A diagram of a state  Description automatically generated | | | | | Tasks continue regular processing after the checkpoint barrier is forwarded  A diagram of numbers and circles  Description automatically generated |
| A diagram of a job management system  Description automatically generatedSinks acknowledge the reception of a checkpoint barrier to the JobManager  Checkpoint is complete when all tasks have ackowledged the successful checkpointing of their state | | | | | |
| Spark vs Flink | | Spark  - Microbatch streaming processing (latency of a few seconds)  - Checkpoints are done for each microbatch in a synchronous manner (“stop the world”)  - Watermark: a configuration to determine when to drop the late events | | | | Flink  - Real-time streaming processing (latency of milliseconds)  - Checkpoints are done distributedly in an asynchronous matter (more efficient -> lower latency)  - Watermark: a special record to determine when to trigger the event-time related results  - Flink uses late handling functions (related to watermark) to determine when to drop the late events | | | |

|  |  |  |
| --- | --- | --- |
| PageRank | Algo for graph mining. Web as a directed graph. Nodes = Webpages. Edges = Hyperlinks.  To rank importance of webpages based on num of in-links. i.e. if A has many pages that has hyperlinks to A, then A is impt  But if we only consider in-links as votes, then malicious user can create a huge num of dummy web pages to link to their 1 page to incr rank  Soln: Make num of votes a page has proportional to its own importance.  Pagerank recursively defines importance of a page based on the importance of the pages linking to it | |
| A diagram of a mathematical equation  Description automatically generatedFor each page j, define its importance/rank as rj. If page j w importance rj has n out-links, each link gets rj/n votes  Simplified pagerank: , where di = num of out-links/out-degree of node i  Flow eqns: ry = ry/2 + ra/2. ra = ry/2 + rm. rm = ra/2. However, no unique soln  To enforce uniqueness, use additional constraint: ry + ra + rm = 1. Then solve to get ry = ra = 2/5. rm = 1/5  Can also model flow eqns using stochastic adjacency matrix M.  If , then M­ji = 1/di else Mji = 0. M is a column stochastic matrix where cols sum to 1  Flow eqn can be written as r = Mr. Where and . Col = start from. Row = end at. | |
| Given r = Mr, can efficiently solve for r using power iteration  1) Initialize **r**(0) = [1/N, …, 1/N]T. 2) Iterate **r**(t+1) = **Mr**(t). 3) Stop when |**r**(t+1) - **r**(t)|1 < , where |x|1 is L1 norm (can use other vector norm)  Intuitive interpretation: each node starts w equal importance (of 1/N). During each iteration, each node passes its current importance along its outgoing edges to its neighbors | |
| Random Walk Interpretation  At time t = 0, surfer starts on a random page. At any time t, surfer is on some page i. At time t+1, surfer follows an out-link from i uniformly at random. Process repeats indefinitely  Let **p**(t) = vector whose ith coordinate = prob that surfer is at page i at time t. So **p**(t) is a prob dist over pages. E.g. [y, a, m] in above graph  For above e.g, **p**(t) when t = 0: [1/3, 1/3, 1/3]. At t = 1: [1/3\*1/2 + 1/3\*1/2, 1/3\*1/2 +1/3, 1/3\*1/2] = [1/3, 1/2, 1/6]. At t = ∞: [2/5, 2/5, 1/5]  Stationary Distribution: as t ∞, the probability dist approaches a steady state, representing the long term probability that the random walker is at each node, which are the pagerank scores  Actually, **p**(t+1) = **Mp**(t). If random walk reaches stationary state: **p**s = **Mp**s | |
| OR r = Mr 1) may not converge. 2) Might not converge to what we want. 3) Results are not reasonable | |
| 1) Some pages are dead ends (have no out-links)  A black line on a white background  Description automatically generated- Random walker has "nowhere" to go to  - Such pages cause importance to "leak out"  [1,0] -> [0,1] -> [0,0] -> … -> [0, 0] | A diagram of a diagram  Description automatically generated2) Spider traps: (all out-links are within the group)  - Random walk gets "stuck" in a trap  - Eventually spider traps absorb all importance  [1/3, 1/3, 1/3] -> [1/3, 1/6, 1/2] -> … -> [0, 0, 1] |
| Random walk w teleports: Common values for are [0.8, 0.9]  For spider trap: At each time step, random surfer can 1) follow a link at random w prob or 2) jump to some random page w prob 1-  If at a dead end, always teleport. For e.g. w spider trap, Change node m to be connected to every node including itself    Spider-traps cause random walker to get stuck. Soln: teleport out of it in a finite num of steps  Dead-ends cause importance to "leak" out of system as matrix not column stochastic. Soln: Make matrix col stochastic by always teleporting when at a dead end | |
| . So same as simplified pagerank and is the teleport term  If any dead ends exist, assume we have preprocess by adding connections from them to every other node  Google matrix A: , where = N x N matrix where all entries are 1/N  PageRank eqn: r = Ar | |
| Problems w PageRank:  1) Measures generic popularity of page. Don't consider popularity based on specfic topics. Soln = Topic-Specific PageRank  2) Use a single measure of importance. Other models of importance. Soln = Hubs-and-Authorities  3) Susceptible to Link spam. Artitificial link topographies created in order to boost page rank. Soln = TrustRank | |
| Topic-Specific PageRank | Random walker has a small prob of teleporting at any step  Teleport can go to: | |
| Main idea: Bias the random walk  - When teleporting, pick a page from a set S = {only pages relevant to the topic} (e.g. Open Directory (DMOZ) pages for a given topic)  - For each teleport set S, we get a diff vector (stable state)  . Then A is column stochastic  This weights all pages in the teleport set S equally, but can also choose to assign diff weights to pages  **Compute as for regular PageRank: - Multiply by M, then add a vector. - Maintains sparseness** | |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | Node/Iteration | 0 | 1 | 2 | … stable | | 1 | 0.25 | .25\*.8 + .2 = .4 | .1\*.8 + .2 = .28 | .294 | | 2 | 0.25 | .25\*.4 = .1 | .4\*.4 = .16 | .118 | | 3 | 0.25 | .25\*.3 + .25\*.8 = .3 | .4\*.4 + .2\*.8 = .32 | .327 | | 4 | 0.25 | .25\*.8 = .2 | .3\*.8 = .24 | .261 |   A diagram of a number of probabilities  Description automatically generatedS = {1}. = 0.8  M =  A =  If S = {1,2,3}, = 0.8. rs = [.17, .13, .38, .3] M = . A = | |
| Create diff PageRanks for diff topics. The 16 DMOZ top-level categories: arts, business, sports, …  Which topic ranking to use? - User can pick from menu. - Classify query into a topic. - User context (e.g. user's book marks, …)  - Can use context of query (e.g. query is launched from a web page talking about a known topic; history of queries) | |
| Graph algos | Features of graph algos: 1) Local computations at each vertex. 2) Passing messages to other vertex  Algos are implemented from the view of a single vertex, performing 1 iteration based on messages from its neighbor  - Similar to MapReduce, user only implements a simple fn, compute(), that describes the algo behavior at 1 vertex, in 1 step | |
| Pregel | Computation consists of a series of supersteps  In each superstep, framework invokes a user-defined fn, compute(), for each vertex (conceptually in parallel)  A diagram of a message  Description automatically generatedcompute() specifies behavior at a single vertex *v* and a superstep *s*:   * Can read messages sent to *v* in superstep *s* -1; * Can send messages to other vertices that will be read in superstep *s* + 1; * Can read or write the value of *v* and the value of its outgoing edges (or add/remove edges)   Termination: - A vertex can choose to deactivate itself. - Is “woken up” if new messages received  - Computation halts when all vertices are inactive | |
| A diagram of a diagram  Description automatically generatedE.g. Computing Max Value  Goal: compute maximum over the values in the vertices in this graph  Approach: Each vertex repeatedly sends it current value to its neighbors (as 'messages'). Then each vertex updates its value to the max over its own value, and all the messages it receives. Process continues until all vertex values stop changing  ***Compute(v, messages):***  *changed = False*  *for m in messages:*  *if v.getValue() < m:*  *v.setValue(m)*  *changed = True*  *if changed:  for each outneighbor w:*  *sendMessage(w, v.getValue())*  *else:*  *voteToHalt()* | |
| 1) Master & workers architecture: - Vertices are hash partitioned (by default) and assigned to workers (“edge cut”)  A diagram of a diagram with scissors and circles  Description automatically generated- Each worker maintains the state of its portion of the graph in memory  - Computations happen in memory  - In each superstep, each worker loops through its vertices and executes compute()  - Messages from vertices are sent, either to vertices on the same worker, or to vertices on different workers (buffered locally and sent as a batch to reduce network traffic)  2) Fault tolerance: - Checkpointing to persistent storage  - Failure detected through heartbeats. - Corrupt workers are reassigned and reloaded from checkpoints | |
| Other Graph Processing Project | 1) Giraph (open-source implementation of Pregel by Facebook)  2) Spark GraphX/GraphFrame (extends RDDs to Resilient Distributed Property Graphs)  - Join Vertex table and Edge table to capture r/s  3) Neo4j (graph database + graph processing) - SQL like interface: Cypher Query Language | |

|  |  |  |
| --- | --- | --- |
| Data Architectures | | A diagram of data storage  Description automatically generatedA diagram of data storage  Description automatically generatedA diagram of data storage  Description automatically generated |
| Database: - Designed to store structured data (i.e. table). - Can be read through SQL queries. - Data adhere to a strict schema - allows DBMS to heavily co-optimize data - storage and processing through an optimized query processing engines  - Very fast computation and strong transactional ACID (Atomicity, Consistency, Isolation, Durability) guarantees on read/write operations  - Online transaction processing (OLTP): traditional databases - Online analytical processing (OLAP): Data Warehouse |
| Data Warehouse:  - central relational repository of integrated, historical data from multiple data sources  Benefits: - store large amt of historical data from diff sources. - High quality data. - Very reliable w strong transactional ACID guarantees  - Modeled w star-schema (fact & dim tables) modeling techniques. - Ideally suited for business intelligence and reporting  But, data warehouses have a hard time addressing the 4 Vs due to:  - extremely expensive to scale out. - Closed formats. - No streaming support. - Limited scaling support  - don't support non-SQL based analytics very well |
| Data Lake: a cost-effective central repository to store data at any scale  - a distributed storage solution, runs on commodity hardware, and easily scales out horizontally  - data is saved as files with open formats. - any processing engine can read and write them using standard APIs  - decouples the distributed storage system from the distributed compute system - Allows each system to scale out as needed - Organizations build their data lakes by independently choosing: 1) Storage system: HDFS, S3, Cloud and etc.  2) File format: Structured: Parquet, ORC; semi-structured: JSON; unstructured formats: text, images, audio, video  3) Computing / Processing engine(s): • batch processing engine: Spark, Presto, Apache Hive  • stream processing engine: Spark, Apache Flink • machine learning library: SparkMLlib, scikit-learn, R  - Pros: 1) Flexibility on choosing storage, data format and processing engines  2) A much cheaper solution than databases -> explosive growth of the big data ecosystem  - Cons: 1) Fail to provide ACID guarantees. 2) Building and maintaining an effective data lake requires expert skills  3) Easy to ingest data but very expensive to transform data to deliver business values  4) Data quality issues due to the lack of schema enforcement 5) Can lead to data swamps |
| Data Lakehouse: system merging both data lake and warehouse (achieved through metadata, caching and indexing layer)  - has flexibility, low cost and scale of a data lake & - has data management and ACID transactions of data warehouses  - Good fit for both BI and ML/AI users. - Especially good match for cloud environment (w separate storage and computing resources) |
| Delta Lake | - the metadata, caching and indexing layer on top of a data lake storage that provides an abstraction level to serve ACID transactions and other management features:  1) Transactional ACID guarantees. 2) Audit History  3) Full DML (Data Manipulation Language) support. 4) Unification of batch & streaming into one processing model.  5) Schema enforcement and evolution.  6) Rich metadata support and scaling  Format: - standard Parquet file w additional metadata  Parquet Files: - column oriented (perform compression on a column-by-column basis). - open source  - self-describing: actual data + metadata (schema & file structure)  Delta Lake Transaction Log (DeltaLog):  - ordered record of every transaction made against a Delta table since it was created  - acts as a single source of truth and tracks all changes made to the table  - Main goal is to enable multiple readers and writers to operate on a given version of a dataset simultaneously  - ACID transactions (spark looks at the transaction log to get the latest version of the table)  - scalable metadata handling. - time travel. - possible transactions (add/remove file; update metadata (change table name, schema, partitioning); set transaction; change protocol; commit info) | |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1. In Spark Structured Streaming, why we need to specify a checkpoint location?  Answer: to save the progress information of a stream query, i.e. what data has been successfully processed. Upon failure, this info is used to restart the failed query exactly where it left off. | | | | | | |
| Text  Description automatically generated with medium confidence  From 12:05 to 12:10, events:   |  |  | | --- | --- | | **Event Table** | | | **sensorID** | **Event Time** | | id1 | 12:06 | | id1 | 12:08 |   And at 12:10 the below result table is triggered:   |  |  |  | | --- | --- | --- | | **Result Table** | | | | **Event Window** | **sensorID** | **Count** | | 11:50-12:00 | id1 | 1 | | 11:55-12:05 | id1 | 1 | | 12:00-12:10 | id1 | 2 | | 12:05-12:15 | id1 | 2 |   From 12:10 to 12:15, we received three more events per below:   |  |  | | --- | --- | | **Event Table** | | | **sensorID** | **Event Time** | | id1 | 11:54 | | id1 | 12:02 | | id1 | 12:13 | | | | | | | Please provide the result table at 12:15.  Answer: watermark = max event time (before 12:10) – watermark delay = 12:08 – 5 min = 12:03  Therefore, intermediate state for 11:50-12:00 is dropped as watermark (12:03) > 12:00. The 11:50-12:00 entry will not be updated, and the rest entries updated accordingly.  **Note:** the intermediate state 11:55-12:05 is NOT dropped as watermark (12:03) < 12:05, there may still be events from 12:03 to 12:05, need to be recorded. Though later, an event 12: 02 (< watermark 12:03) show up, since this entry (11:55 – 12:05) is still active we will record the 12:02 event under this entry (11:55 – 12:05) accordingly.   |  |  |  |  | | --- | --- | --- | --- | | **Result Table** | | |  | | **Event Window** | **sensorID** | **Count** |  | | 11:50-12:00 | id1 | 1 | not update | | 11:55-12:05 | id1 | 2 |  | | 12:00-12:10 | id1 | 3 |  | | 12:05-12:15 | id1 | 3 |  | | 12:10-12:20 | id1 | 1 |  | |
| http://www.newgradiance.com/cru/pictures/otc_pagerank2.gif compute PageRank with a β of 0.7 | | a = 0.3/3. b = 0.7(a/2) + 0.3/3. c = 0.7(a/2+b+c) + 0.3/3  You can understand this term by term: the 0.3/3 comes from the teleport probability of 0.3, which is divided among 3 nodes since it randomly chooses which node to teleport to. For the other terms like 0.7(a/2), note that a splits its PageRank between b and c, while b gives all of its to c, and c keeps all its own. However, all PageRank is multiplied by 0.7 before being sent, as the probability of taking regular steps (i.e. non-teleport) is 0.7.  Solving the equations: we get a = 0.1. Then, the 2nd equation gives b = 0.7\*0.1/2 + 0.1 = 0.135. Finally, since the 3 weights sum up to 1, the remaining weight of 1 – 0.1 – 0.135 = 0.765 must lie with node c (you can check that this also satisfies the last equation). | | | | |
| http://www.newgradiance.com/cru/pictures/otc_pagerank3.gifPageRank with β=0.85 | a = .85c + 0.05 . b = .425a + 0.05. c = .85b + .425a + 0.05  Note that if the question doesn’t ask to solve the equations, you don’t have to solve them. Also note that it is also fine to use the matrix form of the PageRank equations:  The PageRank equation is then , where is PageRank, which could also be written as:  . .  Note that these are equivalent to the equations in the 1st solution (so both are correct answers), since a+b+c=1. | | | | | |
| A picture containing watch, clock  Description automatically generated | | | | 1. Dead ends: {G}, 2. Spider traps: {C}, {E, F, H}, {B, C, E, F, H}, {A, B, C, E, F, H} | | |
| http://www.newgradiance.com/cru/pictures/otc_pagerank4.gifTopic-specific PageRank.  Pages selected for teleport = {1, 2}  teleport probability, (1 - β), is 0.3. | | | r1 = 0.7 r2 + 0.15  r2 = 0.35 r1 + 0.15 r3 = 0.35 r1 + 0.7 r4 r4 = 0.7 r3 | | | |
| **4**. Show pseudocode for the compute() function for the PageRank with teleport (β = 0.85) over vertices algorithm in Pregel / Giraph. Set the initial PageRank value as 1/N (N is the number of vertices), Run 30 iterations and then stop. You can (if you choose) use the functions: getValue(), setValue(), getNumVertices(), getSuperStep(), getOutEdgeIterator(). | | | Compute(v, messages):  if getSuperStep() == 0:  v.setValue(1 / getNumVertices())  if getSuperStep() >= 1:  sum = 0  for m in messages: {sum += m}  v.setValue(0.15 / getNumVertices() + 0.85 \* sum)  if getSuperStep() < 30:  sendMsgToAllEdges(v.getValue() / len(getOutEdgeIterator()))  else:  voteToHalt() | | | |
| 5. On a certain large Spark cluster, Rose creates a data frame named **traceA**, and writes the shown program to process the trace. The **traceA** data frame keeps the logs, and each log entry represents the log information of one web page access, including various fields: **ip**, the IP address of the log entry and **time**, the amount of time of that access.  Line 0: maxSql = spark.sql("""  Line 1: SELECT ip, sum(time) as access\_total  Line 2: FROM traceA  Line 3: WHERE time > 0.1  Line 4: GROUP BY ip  Line 5: ORDER BY sum(time) DESC  Line 6: """)  Line 7: maxSql.collect() | | | | | Depending on whether the dataframe has been in the RAM, if not reading traceA (Line 1-3) may incur potential I/O cost  Depending on the size of data generated from Line 3, Line 4 & 5 can also be the bottleneck as both operations are wide transformation and thus require data shuffling through the network I/O.  If the grouped data after Line 4 is highly skewed (e.g. a super big size of data for certain ip address), Line 5 will have task straggler issue, i.e. certain tasks takes much longer time than the other tasks to complete.  Line 7 may return too many results. We can add LIMIT to the SQL query to limit the number of results. | |