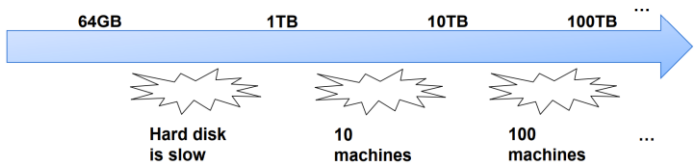
Introduction:

Challenges of Big Data (4 V's):

Volume: The scale of the data

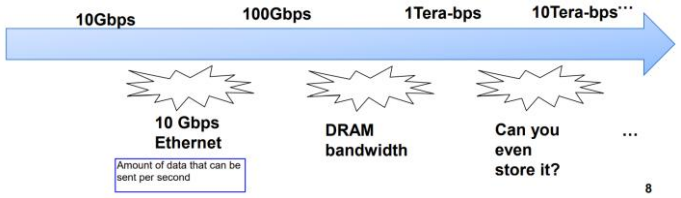


Challenges:

1. Performance
2. Cost
3. Reliability
4. Algorithm Design Complexity

As the size of our data increases, we may need more machines to store them but the reading of the data could be much slower at that point.

Velocity: Speed of new data (streaming data)



Challenges:

1. Performance
2. Cost
3. Reliability
4. Algorithm Design Complexity

If we have high volume of data and it is coming in fast, we may not even have the time to store the data

Velocity: About the format of and structure of the data (Video, chat message, audio, relational data)

Reasons it matters:

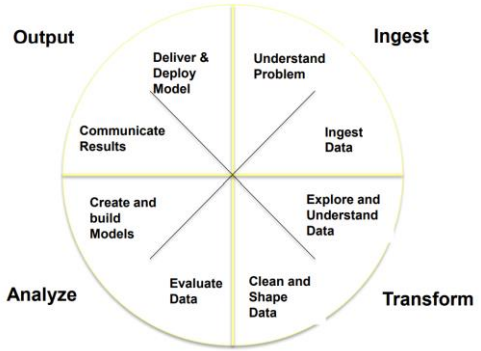
- Not one size fits all
- Data integration
- Multi-modal learning (We may want to combine both audio and video data for machine learning model)

Veracity: Uncertainty of the data

Reasons it matters:

- We could have dirty and noisy data so we need a way to quantify the usefulness of the data
- Data provenance
- Data uncertainty

Data Lifecycle:



Utility Computing:

What:

- Computing resources as a metered service ("pay as you go")
- Ability to dynamically provision virtual machines (Infrastructure to have more resources during peak hours and lesser at non peak hours. Easy to change the scale)

Why:

- Cost: Capital vs Operating Expenses
- Scalability: "Infinite" capacity
- Elasticity: Scale up or down on demand

How providers make money:

- By Economies of Scale, the more computing resources that we use, the cheaper it becomes for the users
- Then once the users are on the cloud, there will be more services introduced and becomes more expensive

Everything as a Service:

Utility Computing: Infrastructure as a Service (Iaas)

- Why buy machines when they can rent cycles? That's why it is good for companies to venture into cloud computing
- **Examples**: Amazon's EC2, Rackspace, Google Compute Engine

Platform as a Service (PaaS):

- Provides hosting for web applications and takes care of the maintenance, upgrades, …
- **Examples**: Google App Engine

Software as a Service (SaaS):

- Just run it for me! (Mostly a subscription service)
- **Examples**: Gmail, Dropbox, Zoom

Principles of Big Data Systems:

Terminologies:

DRAM: Dynamic Random Access Memory. Temporary storage area that holds information. It is much faster to store data here since we do not need to fetch information from the hard drive. The read speeds are much faster for this but the storage capacity is lower

Disk: Hard Drive memory. Traditional computer storage unit. It has a spinning magnetic disk. Benefit over Flash is that it does not suffer inherent deterioration through rewriting data.

Flash: Also a form of hard drive memory. It is much faster as compared to disk, it is small and uses less power, more expensive, less susceptible to unexpected failures

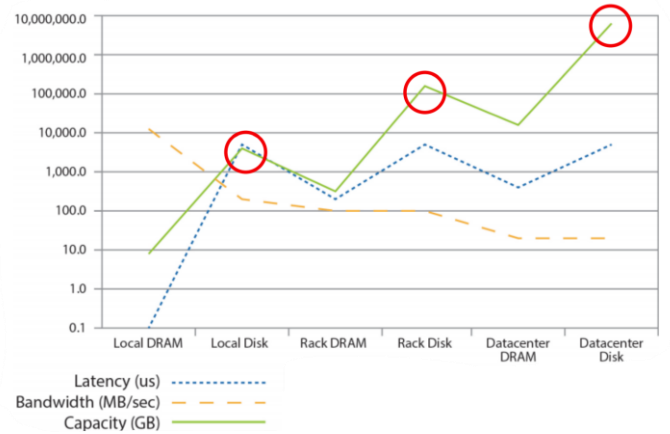
	Bandwidth	**Latency**
Description	**Maximum** amount of data that can be transmitted per unit time (GB/s)	**Time taken** for 1 packet of data to go from source to destination (one-way) or from source to destination back to source (round-trip) e.g. in ms
Size of Data that makes the metric useful	Transferring **large amounts** of data: • Tells us roughly how long the transmission will take	Transferring **small amounts** of data: • Tells us how much delay there will be
Computation	Look at the **smallest bandwidth** through the chain of locations that we have to pass through	**Sum up all the latency** through the chain of locations that we have to pass through

Data Center Architecture:

(***Smallest***) -> (***Largest***)
Server -> Local Rack -> Cluster

		Server	**Local Rack**	**Cluster**
Description		1 Local Machine	Contains 80 Servers	Contains 30 Racks
DRAM	**Storage**	16 Gb	1Tb	30 Tb
	Bandwidth	20 Gb/s	100 Mb/s	10 Mb/s
	Latency	100 ns	300 μs	500 μs
Disk	**Storage**	2 Tb	160 Tb	4.80 Pb
	Bandwidth	200 Mb/s	100 Mb/s	10 Mb/s
	Latency	10 ms	11 ms	12 ms
Flash	**Storage**	128 Gb	20 Tb	600 Tb
	Bandwidth	1 Gb/s	100 Mb/s	10 Mb/s
	Latency	100μs	400 μs	600 μs

Cost of Moving Data around Data Center:



1. ***Storage Hierarchy***:

Capacity increase when we go from Local Server -> Rack -> Cluster.
However, communication costs increases → Latency increases and Bandwidth decreases

- Latency Increases → Need more time to transfer small packets of data
- Bandwidth Decreases → Lesser data can be transferred at one time

Conclusion: Sending data further on a network is costly

2. ***Disk reads are expensive***:

Difference in the order of magnitude is around 3 times, therefore, this is something we want to avoid.
Comparison between the read times of DRAM and Local Disk → Reading from DRAM is always faster no matter what.
Local Disk – Higher Latency and Lower Bandwidth
Conclusion: Read from DRAM as much as possible

Big Ideas of Massive Data Processing in Data Centers:

1. Scale "out", not "up"

Scale out: We try to combine many cheaper machines

- Adding more servers to the architecture to spread the workload across more machines
- **Pros**: Less expensive and there is also a fail safe whereby if one machine goes down, the other machines can still continue running.
- **Cons**: The aggregation of latency could be a factor since we may need to transfer some data across different machines. We also need to ensure that each machine has sufficient processing power, or else it will take very long to process all the data as well.

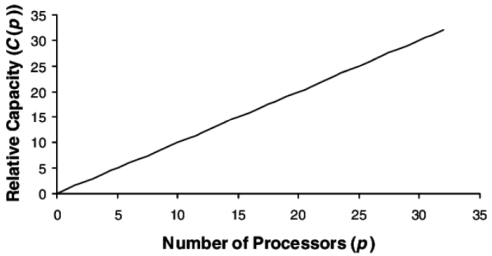
Scale up: We try to increase the power of each individual machines

- Adding further resources, like hard drives and memory, to increase the computing capacity of physical servers.
- **Pros**: Ideally if we scale up, we will be able to handle more data (however, may not always be the case). There is also lower latency since we do not need to aggregate over many machines
- **Cons**: No fail safe, once this machine goes down, we will not be able to process the data anymore. It is also much more expensive to add computing resources to a single machine.

2. *Seamless scalability*

We can scale our operations linearly

Example: If processing a certain dataset takes 100 machine hours, ideal scalability is to use a cluster of 10 machines to do it in about 10 hours



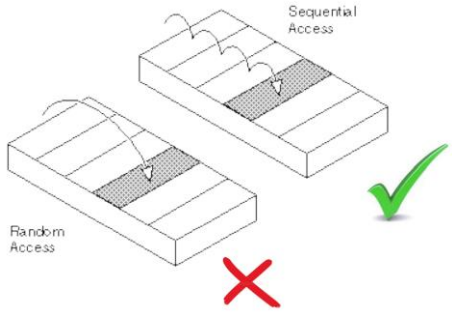
- When we have more machines, the capacity is higher. If our design scales linearly to the number of machines → We can trade more machines with better performance. Assuming in the best case scenario, we can run them in parallel, then we will be able to get an improvement in the performance. (But may not always be the case)

3. *Move processing to the data*

Clusters have limited bandwidth and therefore we should move the program to the machine where the data is stored. Sending data further on a network is very costly

4. *Process data sequentially, avoid random access*

Seeks are expensive, disk throughput is reasonable



- Imagine an array moving around to find something, if we want to run randomly, it is hard to pin point where the value is and therefore it takes a lot of time. To maximise the bandwidth capabilities, we will need to use sequential access.
- For sequential access, we will go from the start to the end
- Note that for random access, there is the whole issue of indexing and we will need to check the indexing to find the item. But we can find the record more easily.

Example: Hard disk

- Random access: 10ms for 4kb = 400Kb/sec
- Sequential Access: 200Mb/sec

Challenges:

1. *Machine Failures*

- There are always potential failure of machines. We may want to scale out but with the possibility of machines failing, then the failure also scales and therefore we need to keep buying more machines.

2. *Synchronisation*

Tough since:

- We don't know the order in which workers run in
- We don't know when workers will interrupt each other
- We don't know when workers need to communicate partial results
- We don't know the order in which workers access shared data

Solution:

- One of the solution is to create a barrier when we have multiple processes. This ensures that every process stops at this point until all processes have reached the barrier.
- Performance will therefore be dependant on the slowest one. It is like asynchronous programming

3. *Programming Difficulty*

Concurrency is difficult to reason about:

- At the scale of datacenters and across datacenters
- In the presence of failures
- In terms of multiple interacting services

Introduction of MapReduce:

Typical Big Data Problem:

1. Iterate over a large number of records
2. Extract something of interest from each *(MAP)*
3. Shuffle and Sort Intermediate Results
4. Aggregate Intermediate Results *(REDUCE)*
5. Generate Final Output

Writing of MapReduce Programs:

1. `map(k1, v1) -> List(k2, v2)` *(MAP)*

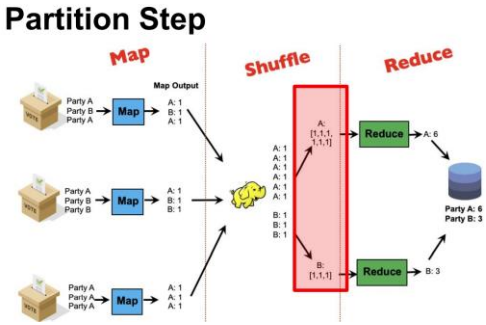
Each map function can result in multiple key value pairs

Output of the map function is the input to the reduce function

2. `reduce(k2, List(v2)) -> List(k3, v3)` *(REDUCE)*

Optional Functions that can be written by the user for speed up:

1. *partition*



Step in between the Shuffle and Reduce

By default, the assignment of keys to reducers is determined by a hash function

Users can implement a custom partition to better spread out the load among reducers

- Can consider this if we believe some keys have too many values and this will cause load imbalance since some of the reducers will need to read in a lot of values. This also causes a bottleneck in the performance

2. *combine*

Step in between Map and Shuffle

Idea is that writing of the map output to the disk is expensive. Therefore, we could probably do some local aggregation first after each map task before writing it into the local disk before the shuffle process

- Note that it is the user's responsibility to ensure that the combiner does not affect the correctness of the final output, whether the combiner runs 0, 1 or multiple times.
    - mean(mean(1, 1), 2) ≠ mean(1, 1, 2)
- We want a combiner that is both Associative and Commutative
    - Associative: a + (b + c) = (a + b) + c
    - Commutative: a + b = b + a
- We also want to make sure that the combiner is meaningful, if the value is a string, we may not want to do any combination

MapReduce Runtime:

1. Handles Scheduling

Assigns workers to map and reduce tasks

2. Handles "data distribution"

Moves processes to data

3. Handles synchronization

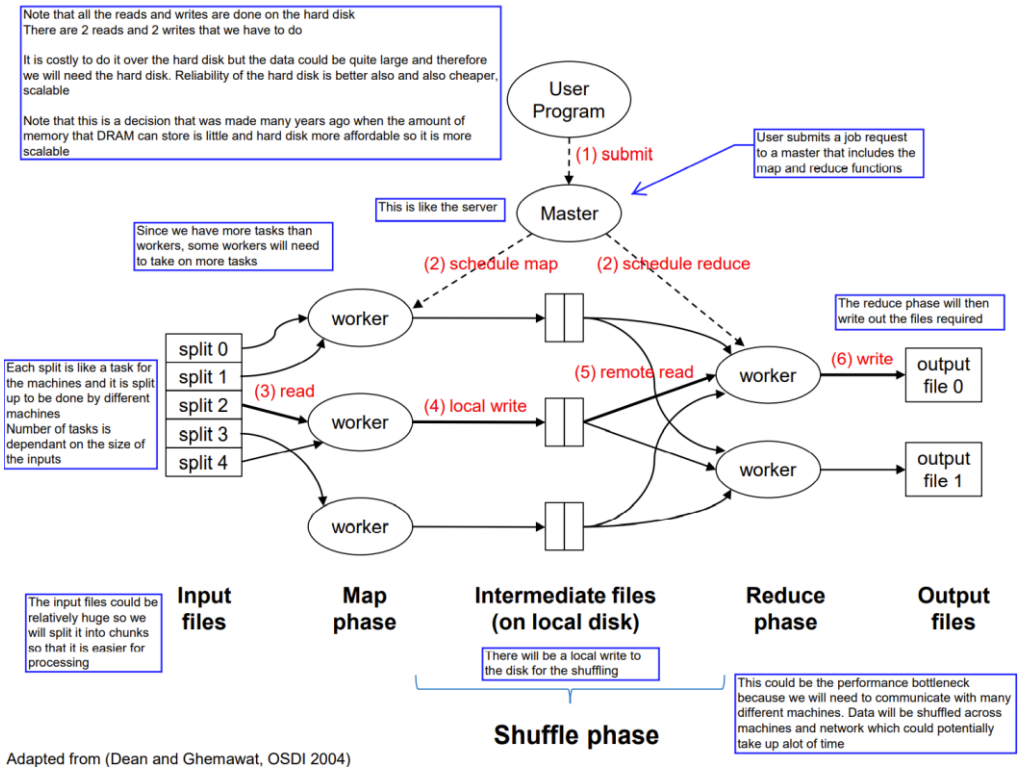
Gathers, sorts and shuffles intermediate data

4. Handles errors and faults

Detects worker failures and restarts

5. Everything happens on top of a distributed file system

MapReduce Implementation:



Adapted from (Dean and Ghemawat, OSDI 2004)

Process:

Note that there are total of 2 reads and 2 writes to be done

1. User submits a job request to the master that includes the Map and Reduce functions
2. The Master schedules the map task and reduce tasks to each of the worker nodes.
    a. Note that each of the worker may have to execute more than one map task
3. The input files are being split into chunks and read in by the respective worker nodes that are supposed to work on a specific map task.
    a. Note that each map task is associated with one input split
    b. The input split is dependant on the input files size. Each of the split will be 128Mb for HDFS systems
4. Each of the map tasks is being executed and the output is a key-value pairing according to the map function specified by the user. The intermediate file output is then written locally onto the hard disk of the mapper that is processing the input split
    a. This local disk write is so that we can do shuffling afterwards
5. Shuffling is done to sort out those intermediate files that have the same keys and they will get partitioned to the same reducer worker node. The reducer worker node reads the intermediate files
    a. Note that this could be the performance bottleneck because we need to communicate with many different machines and the data is shuffled across the network and across different machines which could take up a lot of time
6. The reducer completes the reduce phase using the reducer function that the user specified and outputs the files.

Question: If the size of each split (or chunk) is too big or too small, what are the disadvantages?

- If the chunk is ***too large***: The intermediate result is larger and it needs to be computed on the DRAM so the memory space required for it will be larger. This will cause a lot of contention with other jobs since it uses up a lot DRAM and may cause software failure. We will see out of memory failure a lot of time. Note that these computation will need the DRAM to do then we will store in the hard drive after that. There is also the issue of parallelism, if the chunk size is too large, we will not be able to make use of multiple machines to process the data
- If the chunk is ***too small***: There will be a lot of intermediate files written. Therefore, there will be a lot of disk I/O being utilised for the reading and causing disk I/O performance to degrade.

Terminologies:

Map Task: Basic unit of work. Typically 128Mb.

- At the beginning, the input is broken into splits of 128Mb. A map task is a job that processes one of the splits
- Note that we can have a single machine processing multiple splits.

Worker: A single physical machine

- Can handle multiple map tasks
- Typically, when a machine completes a map task (e.g. split 0), it is then reassigned to another task (e.g. split 3)

Mapper/Reducer:

- Generally refers to the process executing a map or reduce task, not to the physical machines.
- In the above diagram, we have 5 map tasks and therefore 5 mappers but only 3 physical machines

Map Function: Single call to the user-defined `map(k1, v1) -> List(k2, v2)` ***(MAP)***

- Within a single map task, there can be many calls to such a map function
- E.g. within a 128Mb split, there will often be many (key, value) pairs, each of which will produce one call to a map function

Hadoop Distributed File System (HDFS):

- Stores data on local disks of nodes in the cluster
- Start up the workers on the node that has the local data and shift the process to the data instead

Pros:

1. Suitable for large batch of data read and write
2. Suitable for sequential reads and writes

Cons:

1. Not suitable for interactive applications due to high latency
2. Not suitable for concurrent writes from many processes
3. Not suitable for iterative processes (like K-Means)
    a. Hadoop is designed for distributed executions. There will be overhead of runtime scheduling and network oriented designs if we have iterations
    b. Hadoop also needs to repeatedly write to HDFS which is not ideal for every iteration

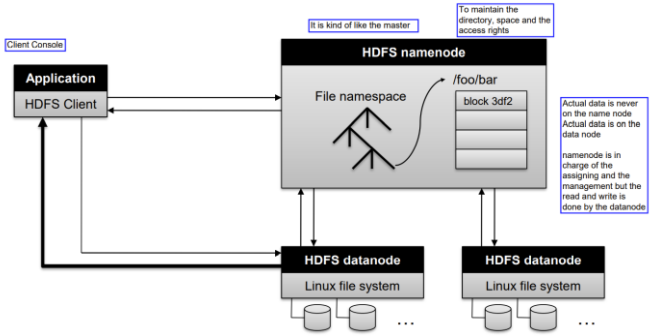
Assumptions:

1. Commodity hardware instead of "exotic" hardware
    a. Scale out and not scale up. Assuming reasonable amount of CPU cores and sizable amount of storage
2. High component failure rates
    a. Inexpensive commodity components fail all the time
3. "Modest" number of huge files
    a. Dozens of Gb
4. Files are write-once, mostly appended to
    a. If allow for modifications of records, we will need to have some sort of indexing to access the records in the middle and it becomes very complicated
5. Large streaming reads instead of random access
    a. High sustained throughput over low latency. Focus is on sequential access so we don't allow for modifications

Design Decisions:

1. Files are stored as chunks across multiple nodes
    a. Fixed size (64Mb for GFS and 128Mb for HDFS)
2. Reliability through replication
    a. Each chunk is replicated across 3+ chunkservers
    b. In a cluster there are 3 copies of the same chunk. (Tradeoff between reliability and performance)
    c. Current Way of Splitting:
        i. Current rack chooses a random machine as the primary one
        ii. Current rack chooses another random one as a back up so that in case anything happens, we have a back up that has fast read speeds
        iii. Last one can be in another rack jus just in case the current rack fails as a whole and we can at least have a back up. (But the read speed may not be that fast)

Architecture:



Question: How to perform replication when writing data?

Answer: Namenode decides on which datanodes are used as replicas. The 1st datanode forwards datablocks to the 1st replica, which forwards them to the 2nd replica and so on.

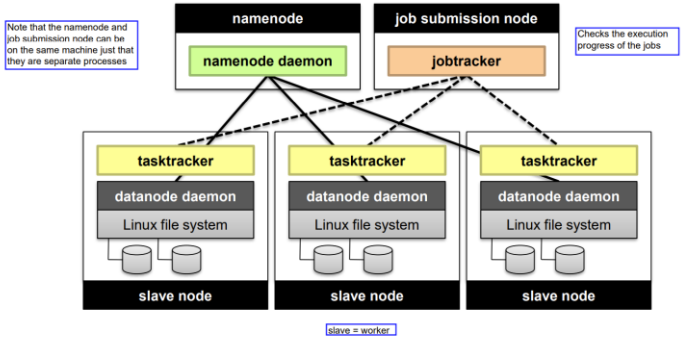
Namenode Responsibilities:

1. Managing the file system namespace
    a. Holds file/directory structure, metadata, file-to-block mapping, access permissions, etc
    b. Coordinating file operations
    c. Directs clients to datanodes for reads and writes
    d. **NOTE**: NO DATA IS MOVED THROUGH THE NAMENODE, IT IS MERELY TO PROCESS THE DATA
2. Maintaining overall health
    a. Periodic communication with the datanode
    b. Block re-replication and rebalancing
    c. Garbage collection

Question: What if the namenode's data is lost

Answer: All files on the filesystem will not be able to be retrieved since there is no way to reconstruct them from the raw block data. Fortunately, Hadoop provides 2 ways of improving resilience, through backups and secondary name nodes

Overall Architecture:



Standard MapReduce Iterative Algorithm Setup:

1. Driver programs sets up MapReduce job
2. Waits for completion
3. Checks for convergence
4. Repeats if necessary

Note:

- With large k, large feature spaces, it could potentially be an issue since the amount of network I/O during shuffle could be high
- Memory requirements of centroids grow over time
- Not ideal to make use of MapReduce for iterative processes (Look at Spark instead)

Performance Analysis of MapReduce

Performance Guidelines for Basic Algorithmic Design:

1. Linear Scalability: More nodes can do more work in the same time
    a. Linear on data size
    b. Linear on computer resources. i.e. When we add more nodes, we want to be able to complete the tasks faster. But we will see that once we hit a certain limit sometimes, even if we add in more computing resources, it may not make the process faster
2. Minimise the amount of I/Os in hard disk and network
    a. Minimise disk I/O: Sequential vs Random
    b. Minimise network I/O: Bulk sends or receives vs Small sends or receives
3. Memory working set of each task/worker
    a. Large memory working set → High memory requirements/ probability of out-of-memory errors. This means that we have too much memory requirements on a single machine's DRAM
4. Load balance among tasks
    a. Load imbalance could cause long execution times / large memory working set

Scalability Analysis:

Assumption: 1 worker can run one map or reduce tasks

Linear Scalability:

- Given W workers, we can run W tasks at the same time

Compute:

1. Max number of map tasks:
    a. Input Size / Chunk Size (**This is what we use to compute**)
        i. Note that the chunk size for HDFS is 128Mb
        ii. This is the max because we will have at most this number of chunks for us to process. So even if we have more machines, we will not be able to capitalise on them since each chunk will go to one machine at most
    b. Will the number of mapper tasks increase linearly as the input size increases
        i. This is basically answering, if increasing the number of machines will increase the performance linearly
2. Max number of reduce tasks:
    a. Number of distinct keys in the reducer (**This is what we use to compute**)
        i. This is the max because each key will go to a reducer. Therefore, we will can at most assign to them this number of reducers. Any more reducers will not be utilised
    b. Will the number of reducer tasks increase linearly as the input size increases

I/O Analysis:

Assumptions: We could assume that we are reading the chunks from the machine that has the chunk. Therefore, there will be no network I/O for the initial read

For the MapReduce task, we have the following I/O Components:

1. Reading input from HDFS: Mainly disk I/O
2. Shuffle and sort: Disk and network I/O
    a. We will need to send the data to other machines for the shuffling and sorting so there is network I/O here
3. Output: Disk and network I/O
    a. Note that there is network I/O here because of replication, output will be in HDFS and it will read from other machines to copy it into their own machine which requires network I/O

Compute:

1. Amount of disk I/O from each Map/Reduce task
    a. Note that for Reduce Task: When we read in the input, normally it will already be in the DRAM since we have done the transfer over the network through the shuffling process. Unless the data is very large that it cannot be kept in the DRAM which will then be stored in the local disk then we read from there, else there isn't much disk reads for Reduce Task
2. Amount of network I/O from each Map/Reduce task
3. Amount of network I/O in shuffle ≈ Amount of intermediate results from map tasks
    a. Note that this is just an approximate because if the reducer runs in the same machine as the one that is mapping, then there is no requirement for the transfer of data across the network

Note that we should look at the following for Map/Reduce/Shuffling steps:

1. Input
2. Intermediate
3. Output

Max working set:

Check the intermediate results

Compute:

- The amount of memory consumption from each map/reduce task
    - Looks at whether there are any data structures that we need to store

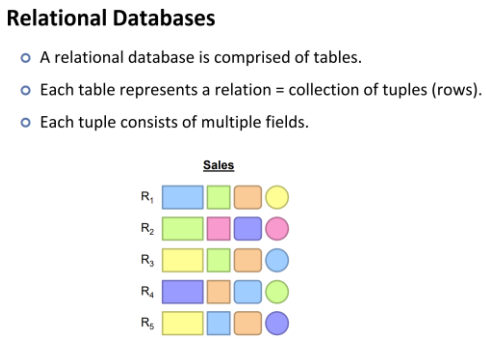
Load Balance:

Avoid unevenly overloading some compute nodes while other compute nodes are left idle

Compute:

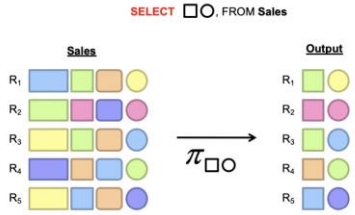
- Worst case analysis for all the map/reduce tasks
- The worst execution time (sensitive to input)
    - Possible Case: When all the keys from the mapper are the same. Therefore, it will go to only 1 reducer and therefore, cause a bottle neck
- Largest memory consumption
    - Possible Case: When we make use of a Hash Table for instance to store intermediate word-count pairs. If all the words are different, the memory size could be very large

MapReduce & Relational Databases:



SQL Operation: Projection

Taking out specific columns from the table



In **Hadoop MapReduce**:

1. ***Map Function***:

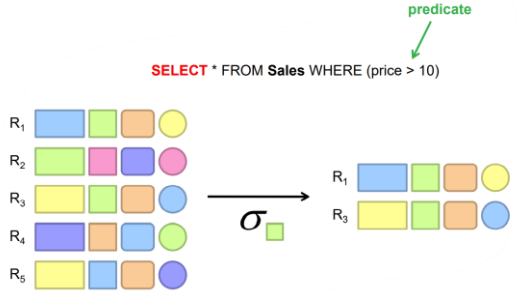
Takes in a tuple (with tuple ID as key), emit new tuples with appropriate attributes (those columns that we require)

2. ***Reduce Function***:

No reducer needed (no shuffle step) since we have already extracted those rows with the relevant columns in the mapper phase

SQL Operation: Selection

Taking out specific rows from the table that satisfies some predicate



In **Hadoop MapReduce**:

1. ***Map Function***:

Takes in a tuple (with tuple ID as key), and emit only tuples that meet the predicate

2. ***Reduce Function***:

No reducer needed (no shuffle step) since we have already extracted those rows that satisfies the condition in the mapper phase

SQL Operation: Group By ... Aggregation

We will group by a certain column and carry out aggregation for those groups along a certain column

Example:

SQL:

SELECT product_id, AVG(price) FROM sales GROUP BY product_id

In **Hadoop MapReduce**:

1. ***Map Function***:

Map over each tuple, emit **<product_id, price>** from the map step

Note that this allows for the GROUP BY product_id since those with the same product_id will go to the same reducer

- We can optimise over here by have a combiner

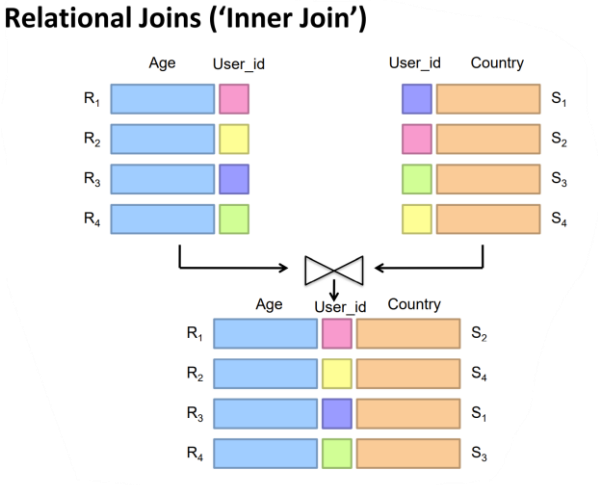
2. ***Reduce Function***:

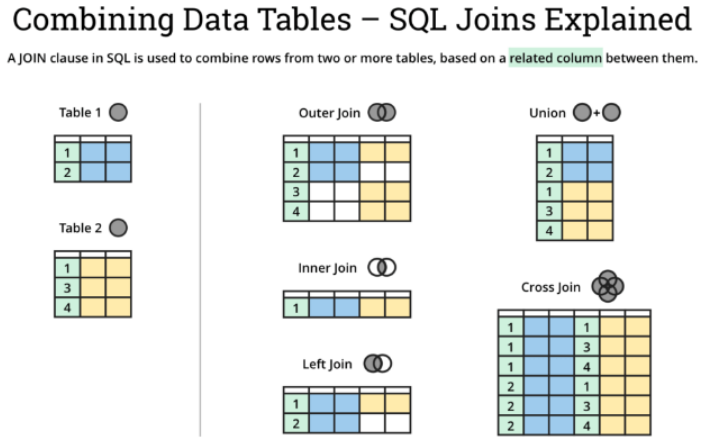
Compute the average in the reducer and output the result

Note that the averages are computed across the groups of product_id which is exactly what is required

SQL Operation: Relational Joins

- This is a basic operation whereby we find similar columns in 2 tables and join them accordingly

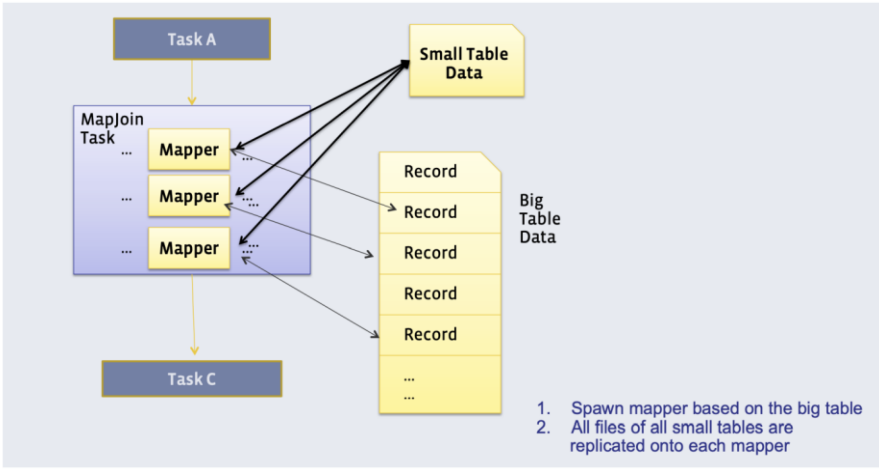




Methods for doing joins in Hadoop MapReduce:

- Note that joins are complex to carry out in MapReduce, there is no "one size fits all" solution

1. Broadcast Join



Note:

- This method requires one of the tables to fit in the main memory of individual servers. Therefore, ideally it should be small
- Can set the number of mappers to be the number of records in the big table

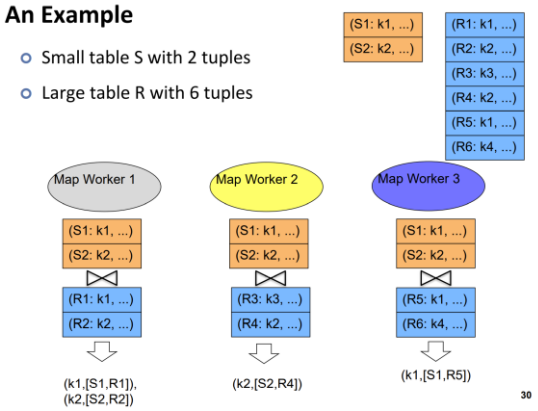
1. All mappers to store a copy of the small table inside their working memory (DRAM)
2. They iterate over the big table and join records with the small table

Map Function:

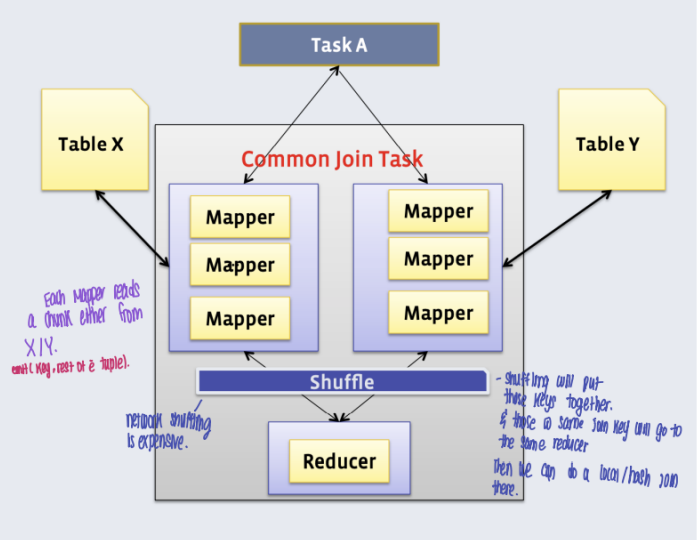
Do a join of the small table with the records in the big table that the mapper is assigned to
Emit the joined rows through the relevant join operations required (inner, outer, etc)

Reduce Function:

No reducer required (no shuffling step) since all the join is carried out in the mapper already



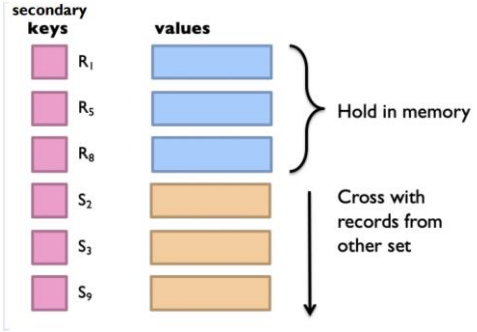
2. Reduce-Side Join



Note:

- Does not require a dataset to fit in memory, but it is slower than Broadcast join since there is a shuffling step required before we do the reducer
- More general use since we can have tables of any size

1. Different mappers operate on each table, and emit records, with key as the variable to join by
2. In the reducer: We can use secondary sort to ensure that all keys from table X arrive before table Y. Then hold the keys from table X in memory and cross them with records from table Y



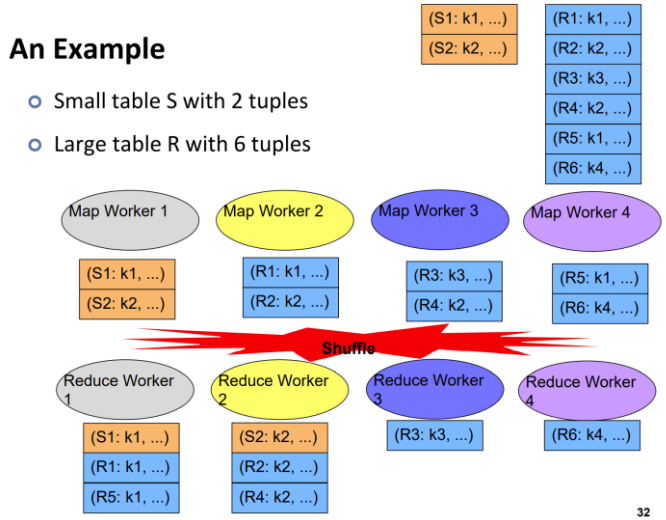
Map Function:

Take in a few records from one of the tables

Emi the records with the key-value output of having key as the variable that we want to join by

Reduce Function:

At each of the reducer, those with the keys that we want to join by will be together. Therefore, we just need to ensure that we do the join across tables that are different and output the joined results.

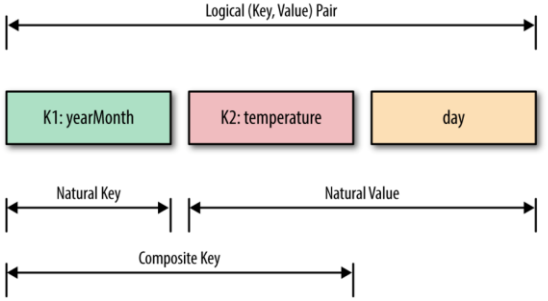


Secondary Sort:

Problem: Each of the reducer's values arrive unsorted. But what if we want them to be sorted

Solution: Define a new "composite key" as (K1, K2) where K1 is the original key ("Natural Key") and K2 is the variable that we want to use to sort

- **Partitioner**: Now needs to be customised to partition only by K1 only, not (K1, K2)
- This will be useful when we do our reducer step to ensure that those values from the first table comes in first and we can just do a cross join with all incoming records when they start from the second table. We will store those from the first table and then just do a cross join and output with those records in the second table which is much more efficient than to check that they are different tables each time a record gets read at the reducer



MapReduce and Data Mining:

Similarity Search:

Distance Metrics:

- The smaller the value, the more similar they are

1. ***Euclidean Distance***:

$$d(a,b) = ||a-b||_2 = \sqrt{\sum_{i=1}^{D}(a_i - b_i)^2}$$

2. ***Manhattan Distance***:

$$d(a,b) = ||a-b||_1 = \sum_{i=1}^{D}|a_i - b_i|$$

3. ***Jaccard Distance***:

$$d_{Jaccard} = 1 - s_{Jaccard}(A,B)$$
$$= 1 - \frac{|A \cap B|}{|A \cup B|}$$

Similarity Metrics:

- The larger the value, the more similar they are

1. ***Cosine Similarity***

Note that for cosine similarity we are only concerned with their direction, it doesn't look at the scale of the vectors.

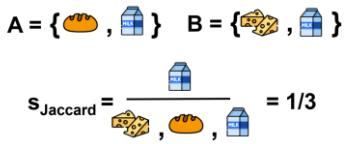
This works because if the 2 vectors are most similar, cos 0 = 1 since their angles are 0

$$s(a,b) = cos\theta = \frac{(a \cdot b)}{||a|| \cdot ||b||}$$

2. ***Jaccard Similarity***

Used between 2 sets A, B

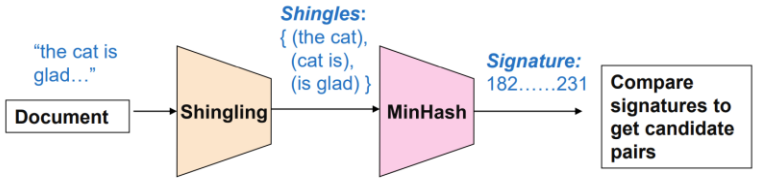
$$s_{Jaccard}(A,B) = \frac{|A \cap B|}{|A \cup B|}$$



Task: Finding Similar Documents:

- ***Goals***:
  - All pairs similarity: Given a large number N of documents, find all "near duplicate" pairs. E.g. with Jaccard distance below a certain threshold
  - Similarity Search: 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 the users both in search results
  - Similar news articles: Cluster articles by the same story

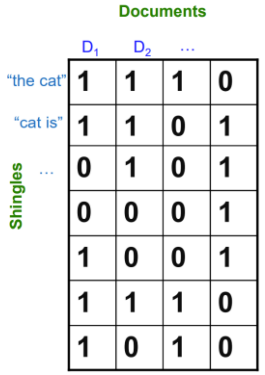
Steps required for Similar Documents:



1. ***Shingling***: Convert documents to sets of short phrases

- k-shingle (or k-gram): For a document is a sequence of k tokens that appears in the document
  - Example: $k = 2$, D_1= "the cat is glad"
  - Set of 2-shingles: $S(D_1)$={"the cat", "cat is", "is glad"}

- **Similarity Metric**: Make use of Jaccard Similarity
  - Each document D_1 can be thought of as a set of k −shingles
  - It can be represented as a matric where the columns represents documents, and shingles represent rows



2. ***Min-Hashing***: Converts these sets to short "signatures" of each document, while preserving similarity
- Note that a "signature" is just a block of data representing the contents of a document in a compressed way
- Documents with the same signature are considered candidate pairs

Motivation for MinHash:
- If we have $N = 1\ 000\ 000$ documents
- To compute the pairwise Jaccard Similarities for every pair of documents it will be:

$$\frac{N(N-1)}{2} \approx 5 \times 10^{11}$$

Arithmetic Series: If $a_n = a_{n-1} + c$, where c is a constant

$$\sum_{i=1}^{n} a_i = a_1 + \cdots + a_n = \frac{n(a_n + a_1)}{2}$$

Geometric Series: If $a_n = ca_{n-1}$, where $c \neq 1$ is a constant

$$\sum_{i=0}^{n} a_i = a_1 + \cdots + a_n = a_1\left(\frac{c^n - 1}{c - 1}\right)$$

If $0 < c < 1$, then the sum of the infinite geometric series is

$$\sum_{i=1}^{\infty} a_i = \frac{a_1}{1 - c}$$

- Which means that computation will be very slow
- We want to have a fast approximation to compare all pairs of documents

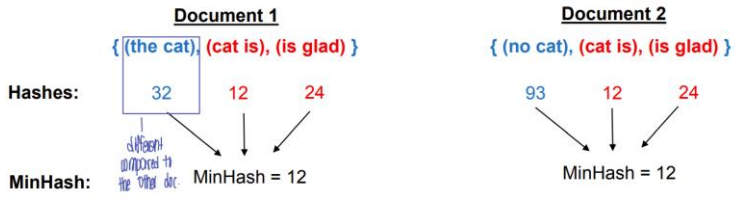
Key Idea: Hash each column C to a small signature $h(C)$, such that
1. $h(C)$ is small enough (~128 bits) that the signature fits in the RAM (It can just be an integer)
2. Highly similar document usually have the same signature

Goals: Find a hash function $h(\cdot)$ such that:
Note that the following similarity function is the Jaccard similarity where we want those that are similar under Jaccard Similarity to have high likelihood to have the same min hash and those that are not similar to not have the same min hash
- If $sim(C_1, C_2)$ is high then with high probability $h(C_1) = h(C_2)$
- If $sim(C_1, C_2)$ is low then with high probability $h(C_1) \neq h(C_2)$

Note that those with the same MinHash does not mean that they have the same value since it could just be mapped to the same value. But those that have the same value will have the same MinHash

Steps: Given a set of shingles (from a document after we have constructed its shingles)
1. We have a hash function h that maps each shingle to an integer
2. Compute the minimum of the hash function output for the set of shingles

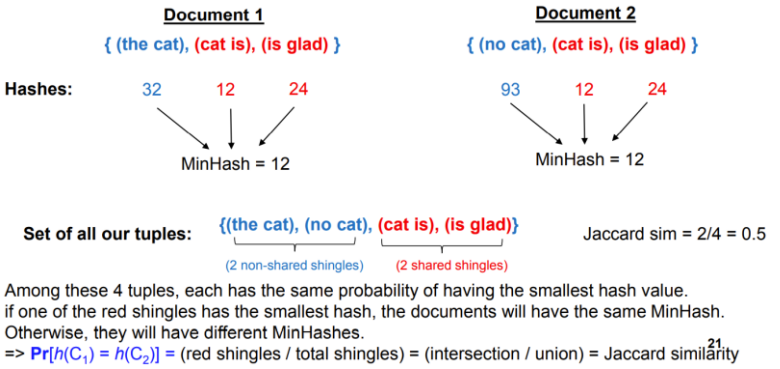
Example: MinHash on 2 sets of shingles



Key Property:
- The probability of that tow documents having the same MinHash signature is equal to their Jaccard Similarity
  - The idea is that the more they intersect, the more likely that they will get the same min hash values since they have a lot of similar shingles so the probability that those shingles are their minimum is high
  - The probability that their minhash is the same is the number of intersections / total of shingles which is the Jaccard Similarity

$$P[h(C_1) = h(C_2)] = JaccardSim(C_1, C_2)$$

Proof:



Note:
1. ***Candidate Pairs***: The documents with the same final signature are "candidate pairs". We can either directly use them as our final output or compare them one by one to check if they are actually similar pairs
2. ***Extension to multiple hashes***: In practice, we usually use multiple hash functions (e.g. N = 100) and generate N signatures for each document. "Candidate Pairs" will then be defined as those matching a "sufficient number" (e.g. at least 50) among these signatures
   a. We will have multiple hash functions and this will help ensure that it is unlikely that there will only be one that is the minimum all the time
   b. This can ensure that we omit those rare occurrences that only have on intersection since the probability over 100 times is very low assuming we use statistically independent hash functions

MapReduce Implementation:

Map Function:

- Read over the document and extract its shingles
  - Note that the value k is user defined
- Hash each shingle and take the min of them to get the MinHash signature
- Emit <signature, document_id>

Reduce Function:

- Receives all the documents with a given MinHash signature
- Generate all candidate pairs from these documents
- (Optional): Compare each such pair to check if they are actually similar
  - Can make use of Jaccard Similarity

Performance Analysis:

1. Scalability Analysis
   a. Map Function: Parallelism is good if documents are good
      i. Max Number of Mappers: Input Size / 128 Mb
   b. Reduce Function: Depends on the min hash distribution, if all the documents are similar, then it will all go to one reducer
      i. Max Number of Reducers: Unique Hash Keys
2. I/O
   a. Shuffle: No of output <signature, document_id> × number of documents (Not a lot required)
   b. Reading of documents: Could be the bottleneck since the input files could be larger
   c. Intermediate Results: Okay
3. Memory Working Set
   a. Similar to tokenizer of WordCount so not much memory required, it is pretty efficient
4. Load Balance
   a. Map: Not much issues
   b. Reduce: If all the documents are similar, then they only have 1 signature and it goes to 1 reducer and it becomes a bottle neck

---

Clustering:

Goals of Clustering:

1. Separate unlabelled data into groups of similar points
2. Should have high intra-cluster similarity and low inter-cluster similarity
   a. High intra-cluster similarity: Distance between points in the same cluster should be small
   b. Low inter-cluster similarity: Distance between clusters should be large

K-Means Algorithm:

1. Initialisation: Pick K random points as centers
   a. Note that k is a hyperparameter that is tuned by the user
2. Repeat the following steps:
   a. Assignment: Assign each point to the nearest cluster
   b. Update: Move each cluster center to the average of its assigned points
   c. Stop if no more changes in assignment

MapReduce Implementation of KMeans:

Version 1:

This MapReduce job performs a **single iteration** of k-means. It receives as input the current positions of the cluster centers (i.e. stars)

```
1: class Mapper
2:   method Configure()
3:     c ← LoadClusters()
4:   method Map(id i, point p)
5:     n ← NearestClusterID(clusters c, point p)
6:     p ← ExtendPoint(point p)
7:     Emit(clusterid n, point p)
1: class Reducer
2:   method Reduce(clusterid n, points [p1, p2, ...])
3:     s ← InitPointSum()
4:     for all point p ∈ points do
5:       s ← s + p
6:     m ← ComputeCentroid(point s)
7:     Emit(clusterid n, centroid m)
```

Network traffic!

Issue: There could be a lot of points being transferred across the network after the Map phase since we have a lot of different cluster ids and the network shuffling could be a lot

Disk I/O Exchange between the mappers and reducers: $O(nmd)$

- Let n = No. of points, m = no. of iterations, d = dimensionality, k = no. of centers
- For each iteration:
  - n points are being emitted, we have d dimensions per point, each of the points we will record their cluster centre which is just a number ⇒ $n \times (d + 1)$
  - m iterations ⇒ $m \times (nd + n) = nmd + nm = O(nmd)$

Version 2:

```
1: class Mapper
2:   method Configure()
3:     c ← LoadClusters()
4:     H ← InitAssociativeArray()
5:   method Map(id i, point p)
6:     n ← NearestClusterID(clusters c, point p)
7:     p ← ExtendPoint(point p)
8:     H{n} ← H{n} + p
9:   method Close()
10:    for all clusterid n ∈ H do
11:      Emit(clusterid n, point H{n})
1: class Reducer
2:   method Reduce(clusterid n, points [p1, p2, ...])
3:     s ← InitPointSum()
4:     for all point p ∈ points do
5:       s ← s + p
6:     m ← ComputeCentroid(point s)
7:     Emit(clusterid n, centroid m)
```

- ***Changes in Version 2***:
  - We have a hash table to keep track of the partial sum instead for each of the cluster id at the Mapper
  - We will output the {cluster_id, partial sum} at the end of each map task
    - Note that this may not be the full sum because some of the points in the same cluster could be a different mapper so the final averaging is done at the Reducer
  - At the reducer, we sum across all the partial sums and take the average over the total number of points

Disk I/O Exchange between the mappers and reducers: $O(kmd)$

- Let n = No. of points, m = no. of iterations, d = dimensionality, k = no. of centers
- For each iteration:
  - k cluster ids are being emitted, we have d dimensions per partial sum of the cluster id, each of the points we will record their cluster centre which is just a number $\Rightarrow k \times (d + 1)$
  - m iterations $\Rightarrow m \times (kd + k) = kmd + km = O(kmd)$
- Note that this is much better because the number of cluster centres is lesser than the number of points that we have to compute

NOSQL

- Rising Popularity due to Volume, Velocity and Variety of data
- Cost of data has been going down as well

Distributed Database:

- **Assumptions**:
  1. All nodes in a distributed database are well-behaved (i.e. they follow the protocol we designed for them; not trying corrupt the database)
  2. Users should not be required to know how the data is physically distributed, partitioned or replicated
  3. A query that works on a single node database should still work on a distributed database

---

Types of NOSQL Systems

Key-Value Stores

Features:

- Stores association between keys and values
  - Keys are usually primitive values that can be queried
  - Values can be primitive or complex; usually cannot be queried
- Improves scalability and efficiency
  - Writing or reading of database is much faster since they only store key-value pairs

Type of Operations:

- ***GET*** – fetch value associated with a key
- ***PUT*** – set value associated with a key
- ***Multi-Get*** (Optional)
- ***Multi-Put*** (Optional)
- ***Range Queries*** (Optional) – Can get a range of values based on their key

Use Cases:

- Small continuous read and writes
- Storing 'basic' information (e.g. raw chunks of bytes), or no clear schema
- Do not need to do complex queries
- When user pages are slightly stale – then eventual consistency is acceptable since they wont be querying the database very often

Cons:

- Cant use this for aggregation

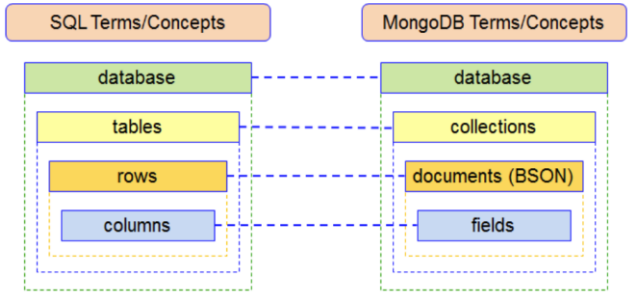
Example of Applications:

1. Storing user sessions
   a. We can make use of the userid to get the session id
2. Caches
3. User data that is often processed individually.
   a. E.g. Patient medical data → (patient_id, info)
   b. However, we cannot do complex queries like "How many patients that took treatment X recovered"

Implementation Issues:

1. ***Non-persistent memory***:
   a. Just a big in-memory hash table
   b. Examples: Memcachaed, Redis (Note that these can also be backed up to the disk periodically)
2. ***Persistent memory***
   a. Data is stored persistently to disk
   b. Example: RocksDB, Dynamo, Riak

Document Stores



- Basically json
- It is a collection of documents where each document can have different fields (columns)

Use Cases:

1. Flexible schema could be beneficial
   a. E.g. special types of vehicles may require different sets of fields
2. Unlike key-value stores, document stores allow for queries based on fields of a document

Type of Operations:

CRUD Operations are allowed

- ***Create***

In MongoDB

```
db.users.insert (          <-- collection
   {
     name: "sue",          <-- field: value
     age: 26,              <-- field: value
     status: "A"           <-- field: value
   }                            } document
)
```

In SQL

```
INSERT INTO users                  <-- table
            ( name, age, status )  <-- columns
VALUES      ( "sue", 26, "A" )     <-- values/row
```

- ***Read***

In MongoDB

```
db.users.find(                     <-- collection
   { age: { $gt: 18 } },           <-- query criteria
   { name: 1, address: 1 }         <-- projection
).limit(5)                         <-- cursor modifier
```

In SQL

```
SELECT _id, name, address  <-- projection
FROM   users               <-- table
WHERE  age > 18            <-- select criteria
LIMIT  5                   <-- cursor modifier
```

- ***Update***

CRUD: Update

In MongoDB

```
db.users.update(                   <-- collection
   { age: { $gt: 18 } },           <-- update criteria
   { $set: { status: "A" } },      <-- update action
   { multi: true }                 <-- update option
)
```

In SQL

```
UPDATE users          <-- table
SET    status = 'A'   <-- update action
WHERE  age > 18       <-- update criteria
```

- ***Delete***

In MongoDB

```
db.users.remove(           <-- collection
   { status: "D" }         <-- remove criteria
)
```

In SQL

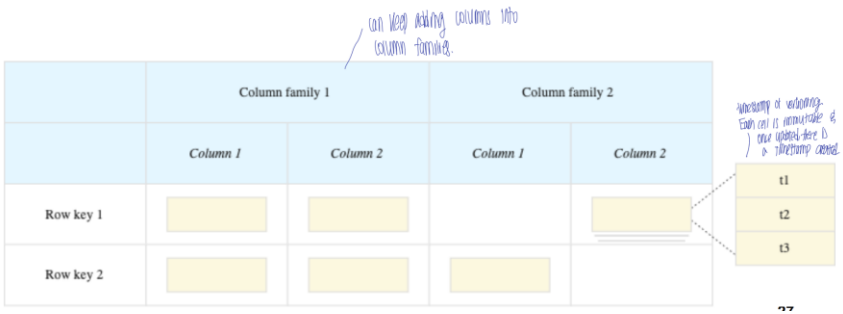
```
DELETE FROM users          <-- table
WHERE  status = 'D'        <-- delete criteria
```

Example of Applications:

- MongoDB

Wide Column Stores:

- Each row describes entities
- **Column Families**: Related groups of columns are grouped together
  - Need to **pre-determine** this

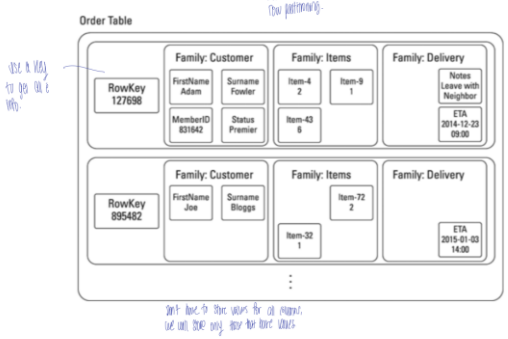


Benefits:

- Sparsity: If a column is not used for a row, it doesn't use the space

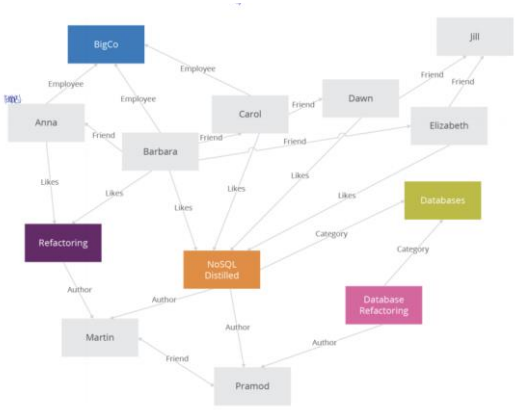
Example of Applications:

- Big Table, Cassandra, HBase



An example of BigTable (an order table)

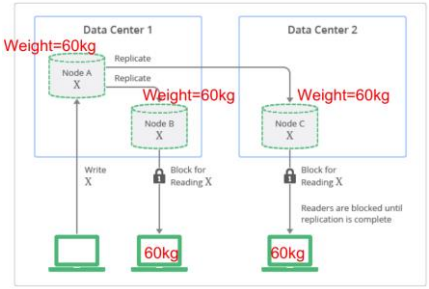
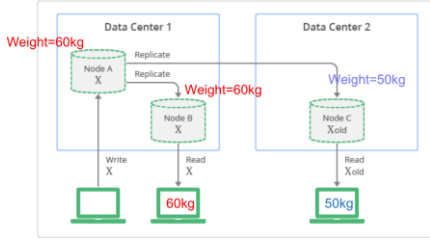
Graph Databases:



- Understand the relationship between different data nodes (entities)

Comparison between Relational DBMS and NoSQL

	Relational DBMS	NoSQL
Guarantees	**ACID Guarantee** 1. **Atomicity**: All changes in one transaction in one transaction either succeed or fail as a complete unit a. Will rollback if there are failures in any single part of a transaction 2. **Consistency**: Database transitions from one consistent state at the beginning of the transaction to another consistent state at the end of the transaction 3. **Isolation**: Concurrent transactions are not affecting each other a. Isolated sequential transactions. When multiple transactions occur, they do not interfere with each other 4. **Durability**: Persistence of committed transactions even under system failure a. Will not change values even during system failure	**BASE Guarantee** 1. **Basically Available**: Basic reading and writing operations are available most of the time 2. **Soft State**: The state of the data could change without application interactions due to eventual consistency a. Note that since we may need some time for the data to be processed at a node, even without user interaction, the state of the data could change b. State of the system is always 'soft' or changing with inputs, until it reaches 'eventual consistency' **3. Eventual consistency** a. Will eventually become consistent (i.e. multiple reads with provide the same result)
State	**Always** in a static state	**Does not** have a static state
Propagation into servers	**Hard** to propagate among many servers	**Easy** to propagate among many servers

Consistency	**Strong Consistency**: Any reads immediately after an update must give the same result on all observers	**Eventual Consistency**: If the system is functioning and we wait long enough, eventually all reads will read the last written value • This means that we may not have the most updated data at times, however, after some time, it will definitely update it
Speed of reading data	Could be **slower** for reading of data because we need to ensure that the replication is done on all observers and that the data read at any time is correct  • This is done through **block reading**: We will block readers from accessing the data until replication is done. • It is an expensive operation since we will lose availability of data	Could be **faster** as we do not need the data to be there at all times. If we give up the ACID constraints, performance becomes much faster  • Note that the users could get outdated data. But if they wait long enough, it will get updated
Use Cases	When the reliability of data being read is important: 1. Banks. We will need the most updated data for the users and we could be okay with the block waiting	
Joins	**Normalisation**: • We do not replicate data that we can join on • Therefore, we do a lot of joins to get the information	1. Some NoSQL databases support joins (later versions of MongoDB) **Denormalization (Duplication)** • Unlike traditional DBMS, we replicate the data instead • Idea is that "storage is cheap" and therefore we can just duplicate the data into many tables so that we don't have to keep carrying out joins • Tables become designed around the queries that we want to receive. This is good when we just need to process a fixed type of queries. (Very fast read speeds)

		• New Problem: If the user changes data for one of the columns, it needs to be propagated to multiple tables and that in itself will take a lot of time
Schema	**Rigid** Schema	**Flexible** Schemas
Language	**Declarative** Language SQL • High level programming	No **Declarative** query language • Query logic (e.g. joins) may have to be handled on the application side, which can add additional programming •
Partitioning of Data		**Horizontal scalability**: Improves speed due to parallelization , if we get more data, we can just split them into more shards, even if each table is large.

Data Partitioning:

1. **Table Partitioning**

Put different tables (or collections) on different machines

Problem: Scalability. If the tables are not balanced, it could cause load imbalance as well and we will not be able to split tables across multiple machines.

2. **Horizontal Partitioning**

Different tuples are stored in different machines (nodes)

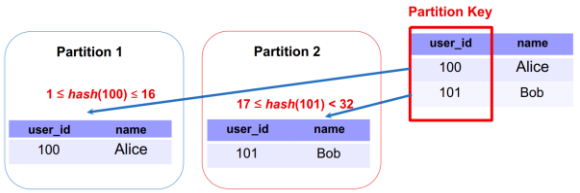
Partition Keys:

- Partition Key (or "shard" key) is the variable used to decide which node each tuple will be stored on: tuples with the same shard key will be on the same node
  - Choice of the partition key: If we need to filter tuples based on a column, or "group by" a certain column, then we can make use of that column as a "partition key". It is really application specific
  - E.g. If we filter tuples by user_id = 100, user_id is the partition key, all user_id = 100 will be on the same partition. Data from other partitions can then be ignored since we don't need data from those partitions ("partition pruning") Time saved by not scanning those tuples

Range Partitioning:

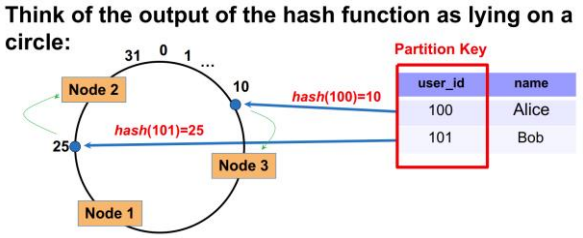
- Split the partition keys based on a range of values
- Will be good when we need range-based queries. Same idea as above, can help with partition pruning
- However, there could be imbalanced shards where some range have a lot of values
- Splitting the range is automatically handled by the balancer that tries to keep the shards balanced

Hash Partition:



- Use hash partition keys and then divide the partitions based on the range of values from the partition keys
- Hash functions automatically helps to spread the partition keys values rather evenly

Solution for having to removing/adding a node: **Consistent Hashing**



Each of the partitions (markers) can be inserted into some part of the circle

To partition:

- Each of the tuples (rows) will be hashed to a hash value. With the hash value on the circle, we will assign that tuple to the node that comes clockwise-after it

Deleting a node:

- Reassign all its tuples to the node clock-wise after that

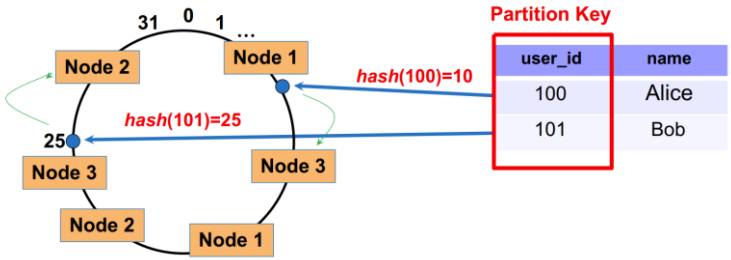
Adding a node:

- Nodes can be added by splitting the current largest node into two, i.e. we just insert the new node in between the range of values for the largest node

Replication:

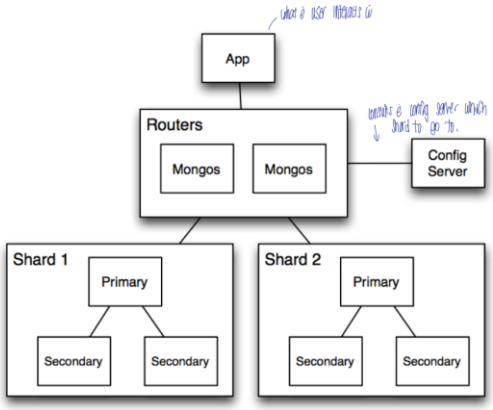
- We can just replicate the tuple in the next few (e.g. 2) nodes clockwise after the primary node to store it.
  - i.e. Just the same as how we assign but use the next available one that has not stored the current value

Multiple Markers:



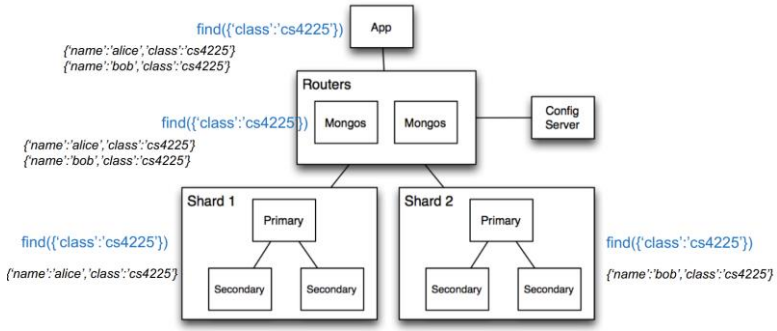
- We can have multiple markers per node spread across the circle. It does not need to be a sequential spread, it can be random and makes the reassign better as well.
- For each tuple, we will still assign it to the marker nearest to it in the clockwise direction
- **Benefit**: When we remove a node, all its tuples will not be reassigned to the same node, there it can balance the node better

Query Processing in NoSQL:



- ***App***: What the user interacts with
- ***Routers (Mongos)***: Handles request from application and route the queries to the correct shards
- ***Config Server***: Stores meta data about which data is on which shard
- ***Shards***: Stores data and run queries on their data

Example of Read or Write Query:



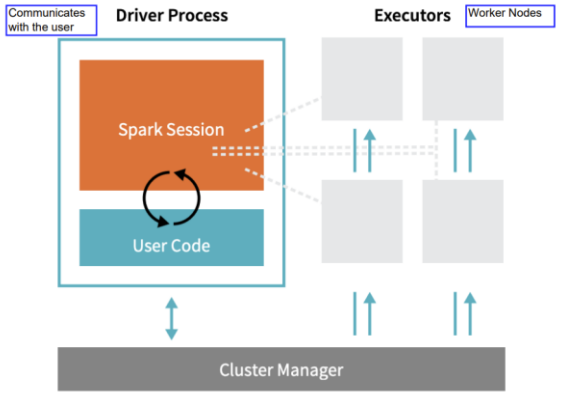
1. Query is issued to a router (mongos) instance
2. With the help of the config server, mongos determine which shards should be queried
3. Query is then sent to the relevant shards
4. Shards run query on their data, and send results back to the mongos
5. mongos merge the query results and returns the merged results

Spark:

Comparison between Spark and Hadoop

	Hadoop	Spark
Memory Storage	**Disk**	**In Memory** • However, if the memory is insufficient, it will store it in disk. Therefore, requiring Disk I/O reads
Iterative Processes	**Does not** perform well Network and Disk I/O Costs: • Intermediate data has to be written to local disks and shuffled across machines which is slow • There is also overhead in scheduling the tasks It has a has a very rigid process that requires the whole shuffling process etc	**Performs well** as intermediate results are stored in memory, making it much faster for iterative processes • As stated above, only when the memory is full, then it spills over to the disk
Concept	MapReduce is a very classic solution which shows that distributed computing works well	Focused on the **computation** that the popular databases are not good at. Since a lot of companies are concerned with the storage, they just make use of the solution solutions from those companies and focuses on the computation and the analytical solution for those companies that needs those complex solutions
Ease of Programming	Need to go through a lot of set up as per what we went through for Assignment 1	**Much simpler syntax** and can work with most programming languages High level programming

Spark Architecture:



Driver Process: Communication with the user

- Responds to user input
- Manages the Spark application
- Distributes the work to the Executors

Cluster Manager (can be Spark's stand alone cluster manager, YARN, Mesos or Kubernetes)

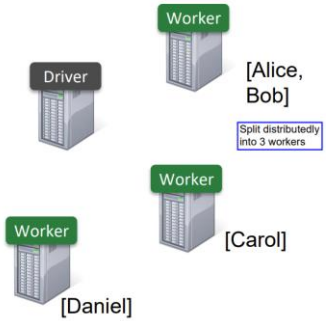
- Allocates resources when the application requests it
- Driver consults the cluster manager to know which Executor to allocate the resources to. Tells us which one is healthy and alive etc.

Executors:

- Runs the code assigned to them and send the results back to the Driver
- Each executor will read data from their own partition and does not read it from the driver. (i.e. the driver tells the executor which part of the data it is assigned to and it reads from there)

Resilient Distributed Datasets (RDDs)

- Achieve fault tolerance through **lineages** (**Does not rely on duplication**)
- Collection of objects that is distributed over machines
- It will read in data from a persistent database (i.e. HDFS) and then we record the transformations that we need to do before we carry out the action (only materializes when there is an action)

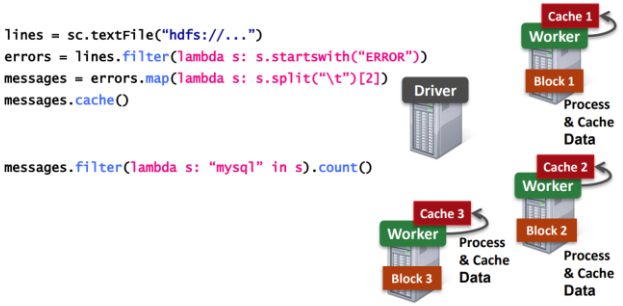


Characteristics:

- **IMMUTABLE**. (They cannot be changed after they are created)
- It will be distributed across various workers in the cluster

Types of operations	Transformations	Actions
Description	Is the way to **transform from one RDD to another** Transformations are **lazy** and it will only be executed when an **action** is called on it • Advantages of **lazy evaluation**: Spark can help to optimise the query plan to improve speed (e.g. removing unneeded operations)	Triggers spark to **compute** a result from a series of transformations When there we need to retrieve all elements of the RDD to the server. (Similar idea to streams in CS2030)
Examples	map, order, groupBy, filter, join, select	show, count, save, collect
Execution	Transformations are done in **parallel** Result is only sent back to the driver in the final step	Actions are also done in **parallel** Result is only sent back to the driver in the final step
	We will only call and **execute all the partitioning steps** etc when an action is called. It is only then **when the driver decides** which executor to call and etc	

Caching:



- We can carry out caching when we know that a **specific RDD will get called multiple times**. (i.e. a particular transformation is a common transformation for 2 RDDs)
- Then we can just cache it into memory so the transformation does not need to happen twice. This **saves up on computation time**.
- Note that if we need to cache the data, it will be cached in memory of the worker node that is handling that partition

Codes:

`cache()` : saves an RDD to memory (of each worker node)

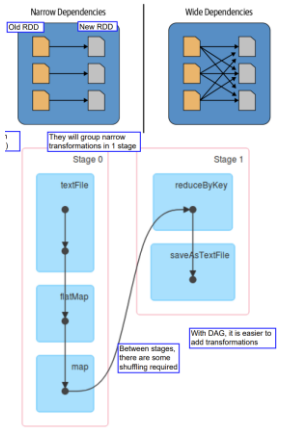
`persist(options)`: can be used to save an RDD to memory, disk or off-heap memory

- This allows for greater flexibility of where we want to store the data ultimately

Choice of whether to cache:

- Note that **memory is limited** as compared to disk and there are other computations that needs to be done in memory. Therefore, storing too many things in cache is not good as well which will cause out of memory errors
- When it is **expensive** and needs to be **re-used multiple times**
- If worker nodes do not have enough memory, they will **remove the "least recently used" RDDs**. Therefore, be careful of the memory limitations since they could remove the values if there is currently no space and when we try to retrieve it, it will not be there.
- We could consider caching when we want to improve efficiency across some repetitive task.

Lineage:



- Internally, Spark creates a graph ("**directed acyclic graph**") which represents all the RDD objects and how they will be transformed
- Transformations constructs the graph and actions triggers the computations on it

Types of Dependencies	**Narrow Dependencies**	**Wide Dependencies**
Description	Transformations where each partition of the parent RDD is **used by at most 1 partition of the child RDD**. Essentially **no shuffling** required	Opposite where each partition of parent RDD is used by multiple partitions of the child RDD **Shuffling is required**.
Examples of transformations	map, flatMap, filter, contains	reduceByKey, groupBy, orderBy
Stages	**Within stages**: Spark performs consecutive transformations on the same machine	**Across stages**: Data needs to be shuffled, i.e. exchanged across partitions, in a process very similar to MapReduce which also **requires writing intermediate data** to the disk • Note that we want to minimise shuffling to improve performance (i.e. Do more narrow transformations and lesser wide transformations)

Fault Tolerance:

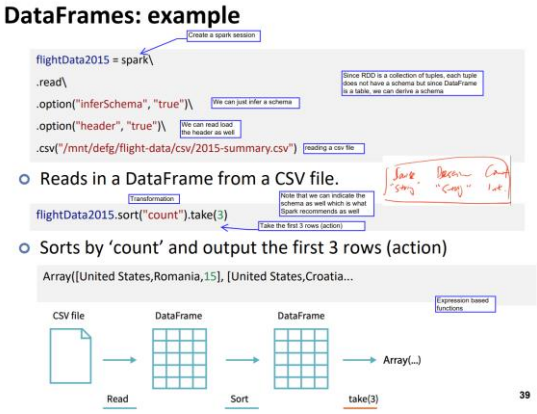
- Spark does not make use of replication (unlike Hadoop MapReduce)
- This is because Spark stores all data in memory and not disk. Memory capacity is much more limited than disk, therefore, to simply duplicate all data will be expensive

Lineage Approach:

- If a worker node goes down, we just **replace it with a new worker node** and make use of the DAG to recompute the data in the lost partition
- Note that we only need to **recompute the lost partition here**
- This allows for **saving up on storage** and **computing**. Since it is not that high frequency for machines to go down, so duplication is not really required and we can save the space for that
- Since we already have the **lineage of how the RDD can be constructed**, we can just make use of that to create the lost partition.
- Note that the data should still be read from a **persistent storage location at the start (like HDFS)**
- With persistent storage location and lineage, it solves this issue of **reconstruction**

DataFrames:

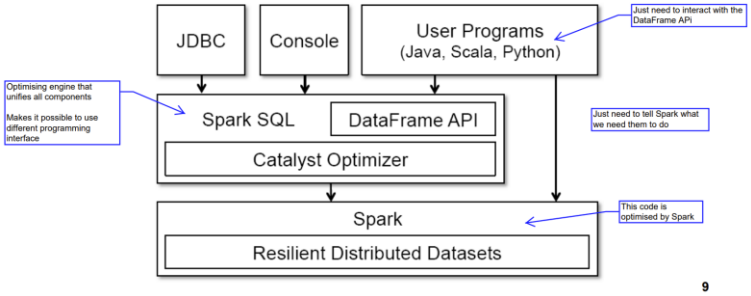
- Represents a table of data, similar to tables in SQL, or DataFrames in pandas
- **Higher level of interface** (declarative programming), we just need to tell the interface what we want and how it is implemented is by the interface
  - RDDs → It is more of a imperative programming whereby we will dictate how each step will be like
  - It has transformations that are more similar to SQL operations
- Note that it is **compiled down to RDD** operations by Spark.
- It is the **recommended interface** to work with Spark since it is very rare that we need to use RDD functions, also Spark can help us to optimise the operations as well
- Has an **optimising engine** that optimises the query and much more friendly interface. Just tell Spark what we want and not how, it is much more efficient than RDD



Transformations in dataframes:

- Note that the DataFrame transformation is the same as RDD transformations. We will still need the starting point to be a **persistent storage (i.e. HDFS)** where we can recover from if anything goes wrong.
- The operations are the same as SQL operations and we just chain them up.
- Each of the transformation functions (groupBy, sort, …) take in either **strings or "column objects"**, which represent columns

Comparison between RDD and DataFrame



	RDD	DataFrame
Programming Style	**Imperative Programming** • We will tell Spark how to compute the query	**Declarative Programming** • Tells Spark what to do instead of how to do it
Intentions	Intentions of the query is **completely opaque** to Spark and therefore, optimisation could not occur • Optimisation is on the onus of the user	**Intentions** of the query is **clear** to Spark • Spark can therefore inspect or parse the query to understand the intention, it can then optimise or arrange the operations for efficient execution.
Structure of data	Spark **will not understand** the structure of the data in RDDs (which are arbitrary Python objects) or the semantics of user functions (which contain arbitrary code)	Code is far more **expressive and simpler** • Uses domain specific language (DSL) similar to python pandas • Use high-level DSL operators to compose the query • Easy for Spark to understand since they are the ones that implements the underlying data structures Note that it is still ultimately compiled to RDDs but this can be optimised
Processing Speed	**Slower** than DataFrame	**Faster than RDD most of the time** since Spark can help us to optimise the operations

Datasets:

- The Dataset `flights` is type safe – its type is the "Flight" class.
- Now when calling `collect()`, it will also return objects of the "Flight" class, instead of Row objects.

- Combination of DataFrames and RDDs. It is DataFrames but with type-safe
- We can do more customised transformations. However, most of the time, the normal DataFrame transformations are sufficient.
- We can transform each row to be a specific class instead

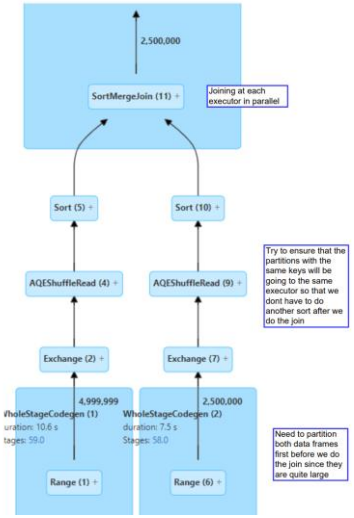
Spark Joins:

1. **Broadcast Hash Join (map-side-only join)**

Smaller datasets will be broadcast to all executors and stored in their own internal memory.
For the larger dataset, each executor will work on a small portion of it
Similar to how the Broadcast join works for MapReduce

- Note that we will need one of the tables to be small

2. Shuffle Sort Merge Join



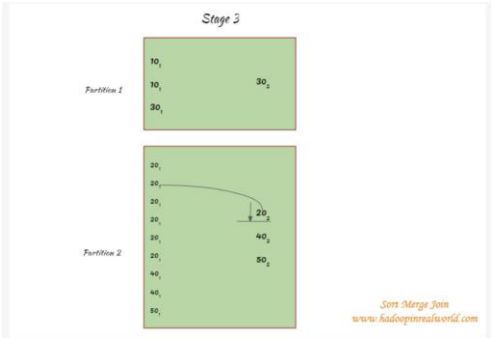
- It is an efficient way to merge 2 large datasets over a common key that is sortable, unique and can be assigned to or stored in the same partition
- All rows within each data set with the same key are hashed on the same partition on the same executor. This is to ensure that we don't have to do another sort after we do the join

Shuffle Phase:

- Data from both datasets are read and shuffled. During the shuffled. After the shuffle operation, records with the same keys from both datasets will end up in the same partition after the shuffle. Note that the entire dataset is not broadcasted with this join. This means that the dataset in each partition will be a manageable size after the shuffle.

Sort Phase:

- Records on both sides are sorted by key. Hashing and bucketing are not involved with this join.
- We do the sorting to ensure that when we iterate through the records to join, once we have a mismatch, we can change records
- A join is performed by iterating over the records on the sorted dataset. Since the dataset is sorted, the merge or the join operation is stopped for an element as soon as a key mismatch occurs.



E.g. In partition 2 where when keys are attempted to match for 20(1), records are iterated through the other side up until 40(2) is reached as the records are sorted, there is no need to iterate through the entire records on the other side

Spark Design Philosophy:

1. Speed

Big data computing solution, therefore it wants speed

2. Ease of use

Dataframes are similar to SQL so easy for programmers to pick up

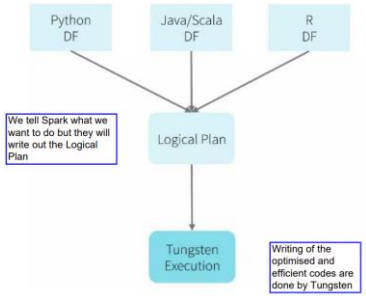
3. Modularity

Separates different functions and is able to combine everything together if required

4. Extensibility

Spark has many different connectors to connect to many different data from different database

Project Tungsten:



- **Objectives**:
  - Substantially improve the memory and CPU efficiency of Spark applications
  - Push performance to the limit of modern hardware
- **How it is done**
  - Memory Management and Binary Processing
  - Cache-aware computation
  - Code generation

Makes Spark very efficient when we have a lot of data to handle

Catalyst Optimizer:

- Takes a computational query and converts it into an execution plan through 4 transformational phases

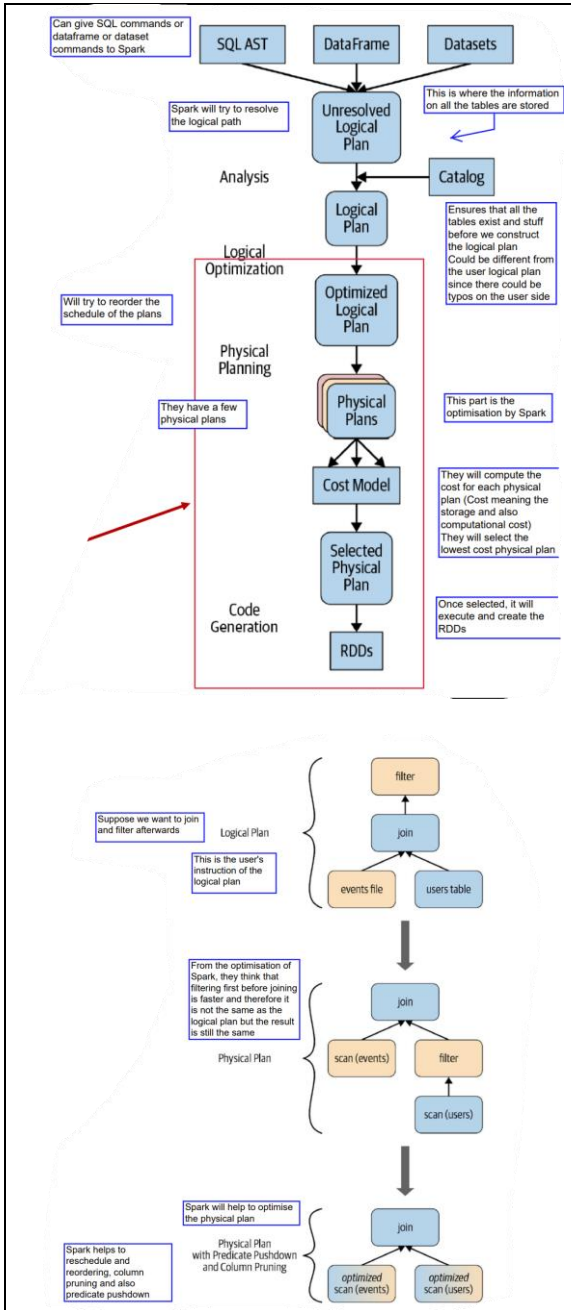
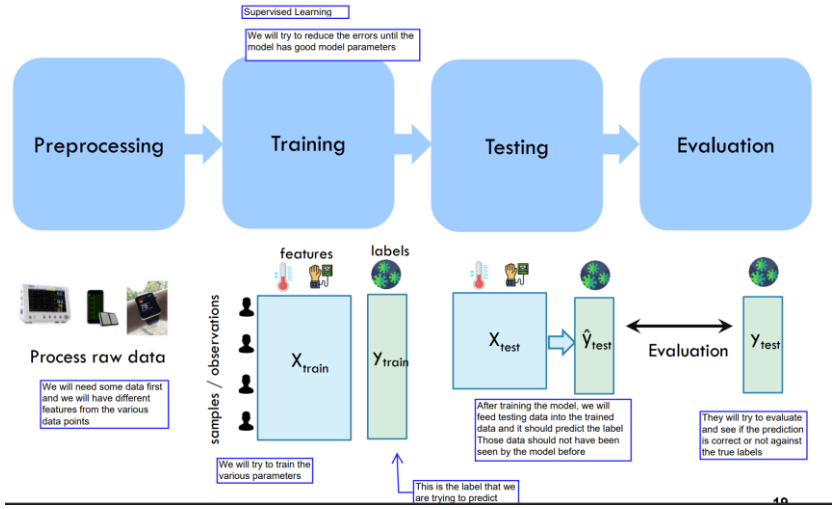


Figure 1 Example of the Physical Plan being different from user's plan

1. **Analysis**
   a. Spark takes in the SQL commands that is issued to it. (This is the Unresolved Logical Plan)
   b. Spark will check the Catalog and ensures that all the tables exist and stuff before we construct the logical plan. Note that this could be different ordering from the user's logical plan since it will be optimised
   c. If there are any issues, like naming issues, it will be raised
2. **Logical optimization**
   a. Spark will try to reorder the plan such the operations will be optimised. As mentioned above, the ordering could be different from what the user stated. However, the results that it generates should be the expected result from the user
3. **Physical planning**
   a. Spark will have a few physical plans that they will try out
   b. They will make use of the Cost Model to compute the cost for each physical plan (Cost in terms of storage as well as computational cost).
   c. They will select the lowest cost physical plan
4. **Code generation**
   a. Once we select a physical plan, it will execute and create the underlying RDD

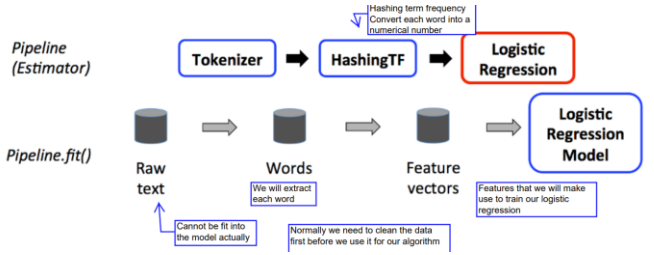
Spark Machine Learning:

Typical Machine Learning Pipeline



1. **Processing**
   a. We will need some data first and we will have different features from the various data points. Thereafter, we will process that raw data into features that we can use
2. **Training**
   a. We will train on various parameters and try to predict a label (Supervised Learning)
   b. Try to reduce the training error until the model has good model parameters. However, we will try not to overfit the model
3. **Testing**
   a. After training the model, we will feed testing data into the trained data and it should predict the label.
   b. Those testing data should not be seen by our training phase before
4. **Evaluation**
   a. We will then try to evaluate and see if the prediction is correct against its true labels (for the testing data)

Pipelines:



- We want to build pipelines so that it includes the data cleaning and the model training so it can be done all at once.
- Note that Spark is also good at this since it can optimise it since we are doing the declarative style of programming here.

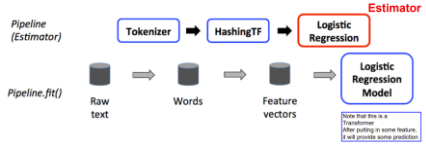
Idea:

- Build complex pipelines out of simple building blocks (e.g. normalization, feature transformation, model fitting)

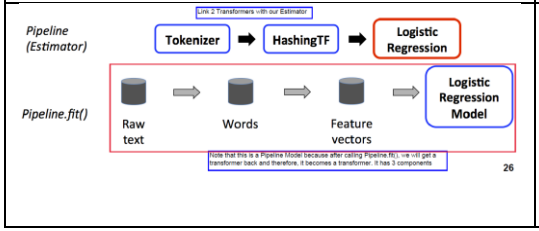
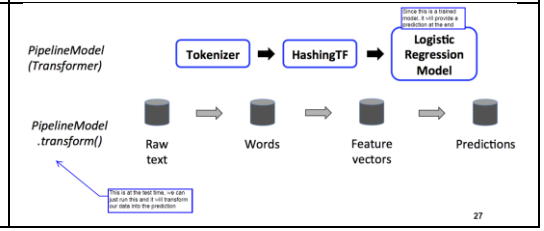
Benefits:

- Better code reuse: Without the pipelines, we will be repeating a lot of code (e.g. between training and test pipelines, cross-validation, model variants, etc.)
- Easier to perform cross validation and hyperparameter tuning

Building Blocks for Pipelines

Type	Transformers	Estimators
Description	For mapping **DataFrames to DataFrames** Note that these transformers will output a new DataFrame which appends their result to the original DataFrame. Therefore, even if we do some transformation of a particular column, it doesn't get deleted, we will just keep updating new columns. (Similar idea to immutable RDDs) • i.e. If we have a fitted model (logistic regression), it will be a transformer that transform a DataFrame into one with the predictions appended	An algorithm which takes in the data and **outputs a fitted model**. For instance, a learning algorithm like the Logistic Regression can be provided with training data and calling fit on it gives us a transformer. The transformer will then if given a testing dataset, transform the DataFrame into one with the predictions appended. 
Examples	One-hot-encoding, tokenization	LogisticRegression
Special method	transform()	fit() → Returns a Transformer

Training vs Testing

Training	Testing
Chains together multiple **Transformers** and **Estimators** to form an ML workflow Pipeline **will be an Estimator** before the fit() function is called. When fit() is called: 1. For transformers, it calls transform() and appends the new columns to the DataFrame 2. For Estimators, it calls the fit() to fit the data and returns a fitted model which a transformer as well. a. Note that fit() process is the longest because we need to train the model based on the data. (Probable bottleneck in performance) At the end of the training phase, we should have a Transformer of an Estimator after it has called the fit() method and will be able to transform the testing data	By using the **output the Pipeline.fit()**, which is an estimated pipeline model (of type Pipeline Model) • It is a **transformer** and consist of a series of Transformers that helps us to clean the data and do the necessary transformations before it is fed into the model for the prediction and the final result will be appended as a column • When the transform() method of the Pipeline Model is called, each of the previous stages' transform() methods are also called
	

Evaluation:

- We can make use of the evaluators to carry out evaluation on the predicted values of the testing data and the ground truth labels of the testing data.

Comparison between MapReduce and Spark

Machine Learning Training each iteration:

1. Feed data into model
2. Get prediction
3. Check for error against the actual label

Feed the error back into the model

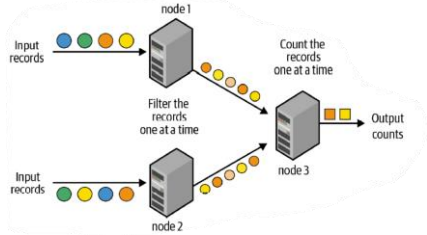
MapReduce	Spark
There are a lot of **overhead cost of rewriting** and reading of data for each iteration through the disk, therefore it is much slower	Saves **the training features into the cache** and it will use that to process the model again for each iteration. Therefore, the **processing for the reading of the initial data is only once**. The rest of the time it will be from memory (note that if memory not enough the it will be from

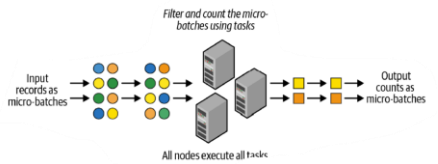
Structured Streaming:

- View the data stream as an unbounded table, new records in the data stream is like new rows appended to an unbounded table

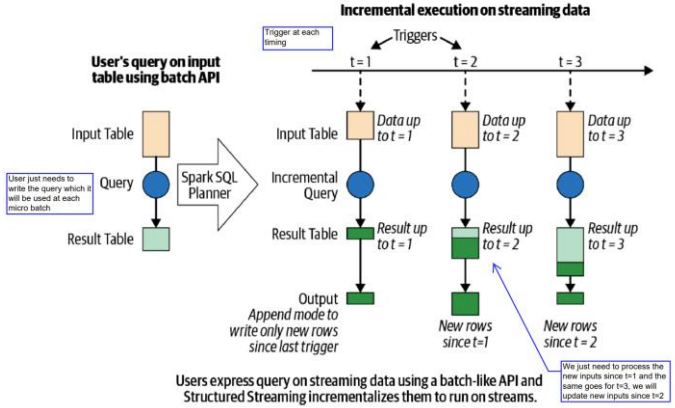
	Traditional Models	Micro-Batch Stream Processing
How they process data	**Processes record one at a time** Each node handles one stream of data. It will filter out the records one at a time and there will be another node that counts the records one at a time.	**Divides the data** from the input streams into micro batches Each batch is processed in the Spark cluster in a distributed manner Small deterministic tasks generate the output in micro-batches • Input each micro batch at a time into the cluster. We can make use of the same API that we have for DataFrames since each batch is like a DataFrame as well
Use Cases	This is what happens normally in batch processing because we have data available all the time	Structured Streaming
Benefits	Can achieve **very low latencies** (e.g. milliseconds)	**Quickly and efficiently recover** from **failures** and s**traggler executors** • If any nodes fail, we can just redirect the input stream to another node instead • Wont have straggler since we can split the whole input stream equally and there wont be imbalance of workload

Cons	**Not very efficient** in recovering from:
	• Node failures
	• Straggler nodes: Nodes that are slower than others
	  ○ Some nodes could be processing a lot of data at once and therefore could be a bottle neck in performance
	  ○ This happens if we have big streams of data coming in at once
		Latencies of few seconds
		• Wont be able to achieve milliseconds, since we break it up into micro batches and we need all to complete before we output
		• Ok for many applications
		• Application may incur more than a few seconds delay in other parts of the pipeline
		There will be a tradeoff between efficiency and latency:
		1. If microbatch is large → Efficiency is high but latency is also high
		2. If microbatch is small → Efficiency is low but latency is also low
		
		Latency is high for large micro batch size because we will need to **wait for more data to complete processing** before we carry on to the next micro batch
		
		Efficiency is high because we complete a **larger batch** each time.



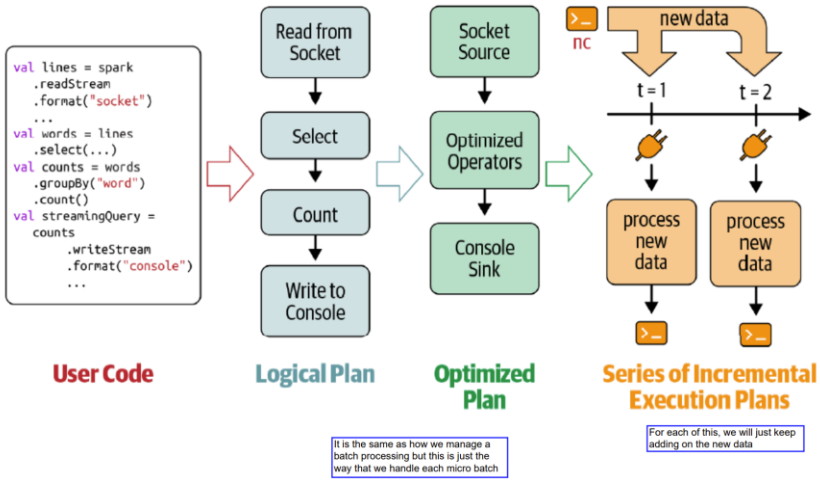


Processing Model:



1. User just needs to **write the query using the batch API** for Spark
2. We will have **different timestamp triggers** for $t = 1, 2, \cdots$
3. We will **incrementally query** on the data based on the data available at that time. We will just need to apply the query based on the new rows since the previous trigger time (i.e. previous micro batch)

Steps to defining a Streaming Query:

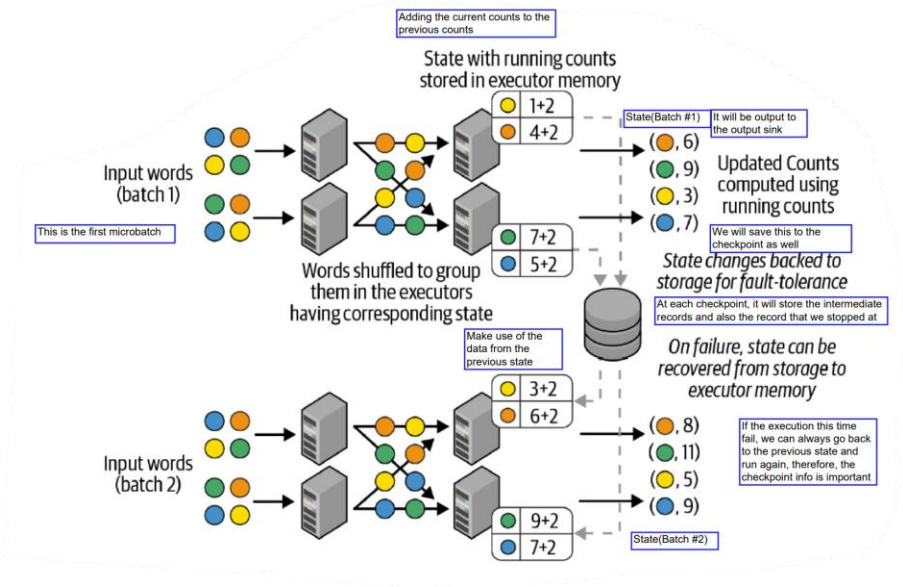


1. Define **Input Sources** (This should be a streaming source)
2. **Transform** Data from the input source (Same way as batch processing)
3. **Define output sink** and output mode (Needs to be a way of storing the streaming data)
    a. Output writing details (Where and how to write the output)
    b. Processing details (How to process data and how to recover from failures)
4. Specify **processing** details
    a. Triggering details: When to trigger the discovery and processing of newly available streaming data
    b. **Checkpoint location**: Store the streaming query process info for failure recovery. Note that since when we are streaming the query, the stream could fail and we need to have a checkpoint location to fall back on
        i. This is useful to **store progress information** about the stream query. Like what are the data that have been successfully update etc and it will be used for us to use to restart the system when any of the queries fail. We will know where to pick it up from
5. **Start** the query

Transformations for Micro-Batch Processing

	Stateless Transformation	Stateful Transformation
Description	Process **each row individually** without needing any information from previous rows (i.e. like narrow transformations)	In every micro-batch, the **incremental plan adds the count** of new records to the previous count generated by the previous micro-batch The partial count needs to be communicated between plans. This is a state that needs to be maintained The state needs to be maintained in the memory of the Spark executors and is checkpointed to the configured location to tolerate failures.
Examples	Projection Operations: select(), explode(), map(), flatMap() Selection Operations: filter(), where()	DataFrame.groupBy().count()

Distributed State Management in Structured Streaming:



Sequence of Events:

1. Read in **first micro-batch**
2. Do **shuffle and sort** so that those with the same keys goes to the same executor.
3. Each executor will keep **track of the state** (in this case it is counts)
4. We will **update the current state of all executors to a persistent storage** (i.e. HDFS)
    a. This is so that on failure, state **can be recovered** from storage to the executor memory
    b. At each checkpoint, we will store the intermediate records and also which record we stopped at so that we know where to go back in case of a failure
    c. If the execution fails the next round, we can always go back to the previous state and run again, therefore, the checkpoint info is important
5. Read in the **second micro-batch**
6. Do **shuffle and sort** so that those with the same keys goes to the same executor
7. The executor will then update its state based on the latest micro-batch
8. Update the latest state to the persistent storage
9. Repeat

Non-time based Stateful Aggregations:

- Aggregations Not Based on Time
    - Global aggregations

```
runningCount = sensorReadings.groupBy().count()
```

    - Grouped aggregations

Because if there is any failure we can just fall back to previous state checkpoint

```
baselineValues = sensorReadings.groupBy("sensorId").mean("value")
```

All built-in aggregation functions in DataFrames are supported. (Can be used as per the normal DataFrame Methods)

- sum(), mean(), stddev(), countDistinct(), collect_set(), approx_count_distinct() and etc

Time-based Stateful Aggregations

- This is **unique to streaming data** since there is a time factor here but it is tough to do so
- We will need to define what is the eventTime window duration

Problem: We could get **late, out-of-order events**. Therefore, we will need a method to deal with such situations

- We may not also get data in sequence and we want to know the time in which the data is captured, not when we receive the data. (***Delay between the time we get the data and the actual time the data has happened***)

```
(sensorReadings
  .groupBy("sensorId", window("eventTime", "10 minute", "5 minute"))
  .count())
```

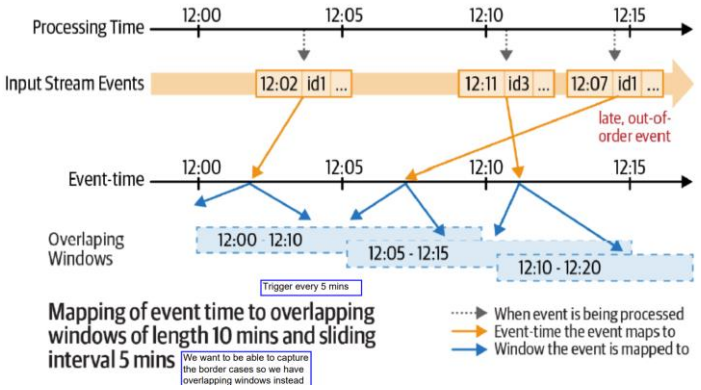


Figure 2 Example of Overlapping Windows

3 Things to take note of:

1. **Tumbling Window**: It is the duration between each of the event window's starting. (i.e. 5mins tumbling windows will be 1200, 1205, 1210 etc)
    a. We will trigger a computation for each of the event time window.
2. **Overlapping Windows**: It is the length of our tumbling windows.
    a. This is to circumvent issues of having event times at the edge of tumbling windows. (i.e. if the tumbling windows are 5 mins, then if the event is at 1205, we don't know where to put it)
3. **Watermark**: It is how long we are allowing for each tumbling window to be updated
    a. It is calculated based on the ***current max eventTime – watermark delay***. Note that the latest eventTime is based on the previous result.
    b. Note that this is **not based on the event time** per say. It is whether the **latest end time of the tumbling window** has exceeded or not.

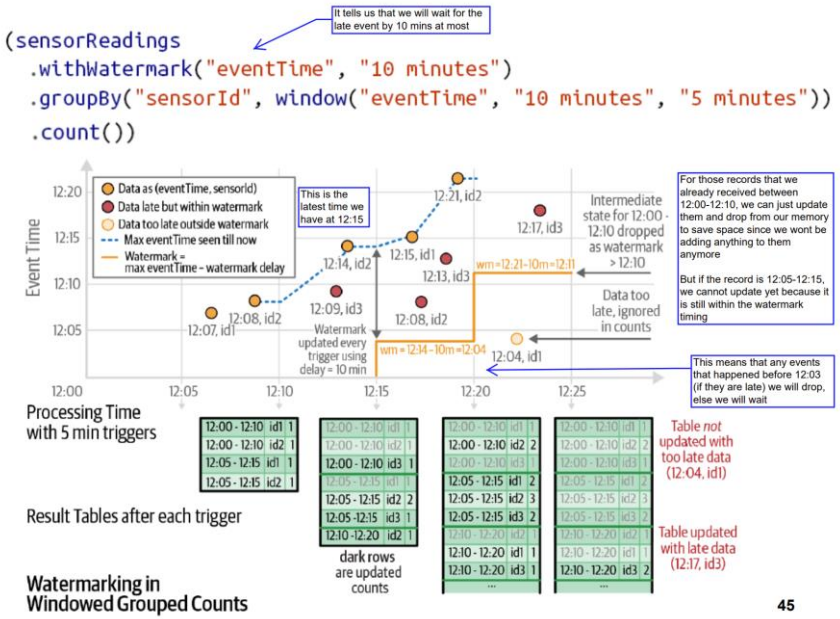


Figure 3 Example of Watermarking

How this works:

1. We will **trigger a computation every 5 minutes** because the tumbling window is 5mins under the groupBy(window("eventTime", 10mins, 5mins))
2. For **each computation**, we will have the current 10 minute windows since we want overlapping windows and it is stated in groupBy(window("eventTime", 10mins, 5mins)) that the windows will be 10 minutes wide
3. We will compute the watermark time based on the previous trigger. So we will look at the previous **latest eventTime** and **– 10 minutes**.
    a. Example at 1225, we have received data during the processing time of 1220 and 1225. We will look at the latest event time at the processing time of 1220 which is 1221 (quite weird but nvm).
    b. The watermark will be 1221-10mins = 1211. We will remove all those eventWindows that are before 1211. This means that all the eventWindows that ended before 1211, like 1200-1210, we will send it as output already and will not update those windows anymore.
    c. However, if we have an entry at like 1206, even though the eventTime is before the watermark time, if there are still eventWindows that are not omitted yet like 1205-1215, we will add it inside there. (**VERY IMPORTANT**, that the watermark time does not omit the eventTime but the eventWindows, so if the eventTime is before the watermark time, it could still be added, depending on the eventWindows)

PageRank

We will model the Web as a graph:

- **Nodes**: Webpages
- **Edges**: Hyperlinks

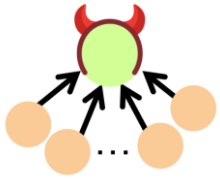
Purpose:

- All web pages are not equally "important". Measuring the importance of pages is necessary so that we can determine which are the pages that are important

Idea:

- We will think of links as votes. A page will be more important if it has more in-links.
- Assuming that incoming links are harder to manipulate. It is easy to create hyperlinks in a website to get to another but it is hard to get other websites to connect to the page.

Problem:



- Malicious users can create a number of 'dummy' web pages, to link to their page, to drive up its rank!
- If a website has many dummy websites being pointed to it, it will drive up the importance of the page

Solution:

- Make the number of 'votes' that a page has proportional to its own importance. Then, as long as the 'dummy' page themselves have low importance, they will also contribute little votes as well.
  - Note that this is a recursive definition. Where links from important pages count more
- If it is an important page, then it has more voting power

"Flow" Model:

Rank/Importance of a page j is given as follows:

$$r_j = \sum_{i \to j} \frac{r_i}{d_i}$$

r_j – Importance of the page j we are concerned with

$i \to j$ – All the pages, i that have an out link going into j.

r_i – Importance of page i which is the page that has an out link to page j

d_i – Number of out links for page i

- Note that we can try to formulate a system of equation to try and solve the equations using the above equations for all the pages. However, there will be 1 redundant equation since it is a cyclical relationship.

We need to also enforce that:

$$\sum_i r_i = 1$$

With the additional constraint above, we will then be able to get a **unique solution**.

Matrix Formulation:

$$r = M \cdot r$$
$$\Leftrightarrow r_j = \sum_{i \to j} \frac{r_i}{d_i}$$

- For **each row, column**. The **row represents where the link** is going to and the column is where the link is coming from. (i.e. M_{ji} means it is going from i to j)
- Note that this formulation works because for each row, we are looking at all the in-links to the node j for instance. We will multiply each of the distributed votes, $1/d_i$ multiplied with their corresponding r_i importance value and sum it to get the importance of j

If page i has d_i out-links

If $i \to j$, then $M_{ji} = \frac{1}{d_i}$ else $M_{ji} = 0$

- Note that M is column stochastic, columns sum to 1. Note that this is because each column represents where the link is coming from, which is how many votes i can give out. It will only give out $1/d_i$ of its votes to each of its outlinks

r : Vector with an entry per page.

r_i – importance score of page i

$$\sum_i r_i = 1$$

Power Iteration Method:

Given a web graph with n nodes, where the nodes are pages and edges are hyperlinks

Power Iteration:

1. Suppose that there are N web pages
2. Initialise $r^{(0)} = \begin{bmatrix} \frac{1}{N} \\ \vdots \\ \frac{1}{N} \end{bmatrix}$
3. Iterate: $r^{(t+1)} = M \cdot r^{(t)}$
4. Stop when $\left| r^{(t+1)} - r^{(t)} \right|_1 < \epsilon$

$$|x|_1 = \sum_{1 \le i \le N} |x_i|$$

Note that we can make use of other distance metrics like Euclidean as well

Stopping Criteria:

- The current one will stop so long as the change is lesser than some threshold, however, this could take some time, so if we do not want to wait, we can consider taking the number of iterations
- Note that we can also stop after n number of iterations. However, we may not know whether the values have converged or not with this approach

Intuitive interpretation of power iteration: Each node starts with equal importance of $\left(\frac{1}{N}\right)$. During each step, each node passes its current importance along its outgoing edges, to its neighbours.

Random Walk

- Idea is basically **a Markov Chain**. Because when we take the transition matrix multiplied with the current probability. The current probability is like the current importance and the transition matrix is the probability of going from each node to another which is the same idea as a M matrix

Stationary Distribution: As $t \to \infty$, the probability distribution approaches a "steady state" representing the long term probability that the random walker is at each node, which are the Page Rank scores

Formula:

$$p(t+1) = M \cdot p(t)$$

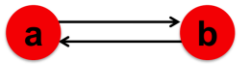
$p(t+1)$ – Probability of the surfer at time $t+1$
M – Transition Matrix
$p(t)$ – Probability of the surfer at time t

Note that once we reach a stationary distribution:

$$p_s = M \cdot p_s$$

Possible Problems with Convergence

1. Does not converge



$$r_j^{(t+1)} = \sum_{i \to j} \frac{r_i^{(t)}}{\mathrm{d_i}}$$

- Answer: not always. Example:

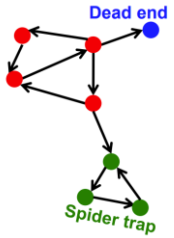
r_a	=	1	0	1	0
r_b		0	1	0	1

Iteration 0, 1, 2, …

If we set the initial value to be 0 and 1, it will oscillate and never converge

If we set the value to be 1 and 0 then it will just keep ding dong between the 2

2. Does not converge to what we want



- **Dead ends**

There are ***no out-links*** for this. It has an in link but no out-link which means once we hit this node, we no longer can move anymore
Random walk does not have anywhere to go
Importance will leak out and eventually become 0

- **Spider traps**

All ***out-links*** are within the group (It is a few nodes whereby once we into this group, it will be stuck within it forever)

- Note that this may not just be the smallest set, we can keep expanding out and it could become bigger also. Check Tutorial 4 Question 1.

Random Walk ***gets "stuck" in a trap***
Spider trap will absorb all the importance (***Stationary distribution***)

Solution for Dead Ends and Spider Traps:

- **Teleport**
- At each time step, the random surfer has 2 options:
  - With probability β, follow a link at random. This will just be the normal page rank algorithm where it follows the out-links
  - With probability $1-\beta$, jump to some random page. When it chooses to jump to another link, it will have equal probability of jumping to any other node including itself.
  - Common value of $\beta = \sim 0.8\ to\ 0.9$

How it solves the 2 problems:

1. **Dead ends**
   a. The problem with dead ends is that it makes the matrix M not column stochastic since it does not have any probability of going to any other node. Therefore, the sum of the column is 0 instead of 1. This will be a preprocessing step
   b. For this, we will make it always teleport at a dead end. (i.e. it will be $1/N$ for the column of a dead end)
2. **Spider Traps**
   a. We will not get stuck in a spider trap by teleporting out of it in a finite number of steps
   b. There is still a large probability of following the link but we have a small probability of teleporting out

Equation:

$$r_j = \sum_{i \to j} \beta \frac{r_i}{d_i} + (1-\beta)\left(\frac{1}{N}\right)$$

- First term is the **probability of following the links** so if they follow the links, the probability of getting to this node will be the previous cumulative importance
- Second term is the **probability of teleporting** to the current node. From any other node, we will need to get the probability to teleport then it is a random chance to reach this node

If any **dead ends exist**, we assume that we have preprocessed **by adding connections from them to every other nodes**

To solve the equation:

1. Write out the transition matrix and also the matrix for the teleporting (which is basically all $1/N$)
2. If it is a dead end, do note to make the column just teleporting for the transition matrix as well
3. Then we can compute the matrix of $r = \beta M + (1-\beta)N$
4. Solve for each of the r using the system of equations
5. Make use of $\sum_i r_i = 1$ as well

Issues with Page Rank:

1. **Measures generic popularity of a page**
   a. Doesn't consider popularity based on specific topics
   b. Solution: Topic-Specific PageRank
2. **Uses a single measure of importance**
   a. Other models of importance. May need different features as well to measure importance
   b. Solution: Hubs-and-Authorities
3. **Susceptible to Link spam**
   a. Artificial link topographies created in order to boost page rank. (Could still have security issues)
   b. Solution: TrustRank

Topic-Specific PageRank

- **Idea**: Bias the random walk to those topic-specific set of "relevant" pages (teleport set)

When the random walker teleports, it picks a page from a set S.
Note that when we teleport, we only teleport to "relevant" pages
S contains only pages that are relevant to the topic

- E.g. Open Directory(DMOZ) pages for a given topic/query
- For each teleport set S, we get a different vector r_s

Matrix Formulation:

$$A_{ij} = \begin{cases} \beta M_{ij} + \dfrac{(1-\beta)}{|S|} & if\ i \in S \\ \beta M_{ij} + 0 & otherwise \end{cases}$$

- Note that A is stochastic
- Compared to the normal Page Rank, when we teleport, we will check if it is in the topic-specific set. If it is in then we will get the probability of teleporting multiplied by an even split of probability for going to any of those topic-specific set through teleporting
- We will weight all pages in the teleport set S equally
  - Could also assign different weights to pages
- Compute as for regular PageRank:
  - Multiply by M (which is the transition matrix)
  - Maintains sparseness (Note that the addition is still a sparse matrix if we cannot teleport, it will be recorded as βM_{ij}

Effects of β and $|S|$

$\uparrow |S| \rightarrow$ ***Increasing the topic set***, the value of the ***original single node*** in the topic set will ***decrease*** because we are sharing the teleporting probability with the rest of the topic set

$\uparrow \beta \rightarrow$ As ***we increase β***, we see that the ***values in the topic set decrease***. This is because increasing β means that we are decreasing the teleporting chances. Therefore, there will be greater influence from the out-link transitions

Finding out the Topic Vectors:

1. Create different PageRanks for different topics
    a. The 16 DMOZ top-level categories: arts, business, sports, …
2. Which topic ranking to use:
    a. Users can pick from a menu
    b. Classify query into a topic
    c. Can use the context of the query
        i. E.g. Query is launched from a web page talking about a known topic. "basketball" followed by "Jordan"
    d. User context, e.g. user's bookmarks

Implementations of PageRank:

- Common Features of graph algorithms:
    - **Local computation** at each vertex. We only need information from the incoming vertices
    - **Passing messages** to other vertices. Only have to pass information to its downstream neighbours
- **Think like a vertex**: Algorithms will be implemented from the view of a single vertex, performing one iteration based on messages from its neighbour.

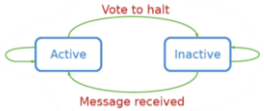
Functions for the user to implement:

- User only **has to implement compute(),** that describes the algorithm's behaviour at one vertex in one step.
- The framework will **abstract away the scheduling/implementing details**. This is because all the rest of the details is the same whereby we just need to pass information to its downstream neighbours and receive information from the its upstream neighbours. We just need to know how to update

Pregel: Computational Model

- Each computation consists of a series of super steps
- In each super step (this is like one epoch), the framework will invoke a user-defined function, compute() for each vertex (conceptually in parallel)
- compute() specifies the behaviour at a single vertex v and at a super step s
    - It can read messages sent to v in superstep $s-1$ (Read messages from the previous super step)
    - It can send messages to other vertices that will read in superstep $s+1$. (Send message to the downstream neighbours and they will use in the next iteration)
    - It can read or write the value of v and the value of its outgoing edges (or even add or remove edges)

Termination criteria:

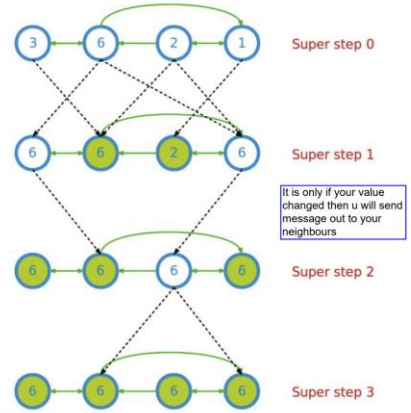


- A vertex can **choose to deactivate itself**. If it keeps receiving a message and they are not changing, then they can deactivate themselves
- Is **"woken up" if a new message** is received. (Can be reactivated again)
- **Computation halts** when **all vertices are inactive**. (This will mean that all values have converged to a certain value)

Example:

```
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()
```

Vote to halt is basically just an indicator to see if it has been changed the current round. If all did not change the previous round then we can terminate since it has converged



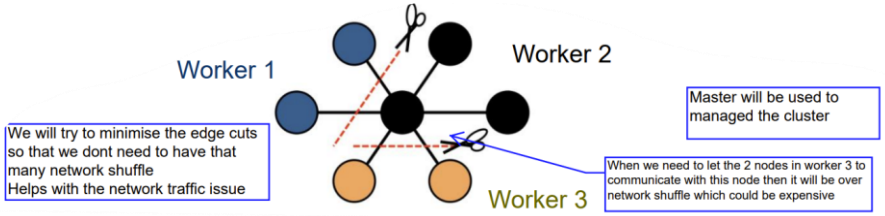
It is only if your value changed then u will send message out to your neighbours

It is only here whereby all didnt change and therefore, all halted and the algorithm terminates

65

Note that we will record ourselves as changed and we will send the message then. Else, if we do not have any changes to our values, we do not send out values. Once all have voteToHalt(), it means that all did not change and we can terminate

Pregel: Implementation



- **Master and workers architecture**
  - ***Master***: Used to manage the cluster
  - ***Workers***: Handles each of the partitions and maintains the state of its portion of the graph in memory.
    - Similar to Spark, all calculation is done in memory. Since PageRank is iterative, then we dont want to store it in disk and keep incurring disk I/O reads.
    - In each super step, 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
      - Sending to different workers, it will buffer locally first and send as a batch to reduce network traffic. This is because network shuffle is expensive so we just try to do it in one shot
- Vertices are ***hash partitioned*** (by default) and assigned to workers through (edge cuts)
  - Try to minimise the edge cuts so that we wont have to transfer the messages through vertices of different machines

Fault Tolerance:

1. Checkpointing to **persistent storage** (i.e. HDFS).
   a. This is because all of the computations are done in memory and we will want to store it into somewhere that is more persistent and in case of any faults, we can retrieve it
2. Failure detected through heartbeats
3. Corrupt workers are reassigned and reloaded from checkpoints

Pregel with PageRank:

- The **compute()** function is basically the PageRank update algorithm depending on what type we require
- We will repeat the **compute() step till superstep** reaches some iteration number (e.g. 30) or we can just repeat until $||_1$ is within some threshold

```
class PageRankVertex : public Vertex<double, void, double> {
public:
 virtual void Compute(MessageIterator* msgs) {
  if (superstep() >= 1) {
   double sum = 0;
   for (; !msgs->Done(); msgs->Next())
    sum += msgs->Value();
   *MutableValue() = 0.15 / NumVertices() + 0.85 * sum;
  }

  if (superstep() < 30) {
   const int64 n = GetOutEdgeIterator().size();
   SendMessageToAllNeighbors(GetValue() / n);
  } else {
   VoteToHalt();
  }
 }
};
```

Compute sum of incoming messages

PageRank update

$$r_j = \sum_{i \to j} \beta \frac{r_i}{d_i} + (1-\beta)\frac{1}{N}$$

1- Beta values
We will have 15% chance that we do teleport

Send outgoing messages

Stopping condition is the number of super steps
We can set another stopping condition to have the change in the updates are lesser than a certain threshold then it terminates

Stop after fixed no. of iterations

Note: this algorithm implicitly assumes that the nodes' values (*MutableValue()) have been correctly initialized based on the initialization scheme that the user wants. In practice, you would generally do the initialization during superstep 0.

Other Graph Processing Projects:

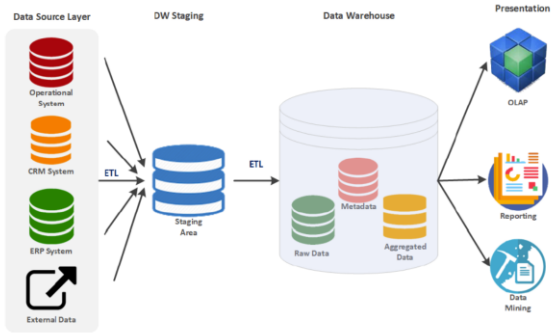
1. **Giraph**
   a. An open-source implementation of Pregel by Facebook
2. **Spark GraphX/GraphFrame**
   a. Extends RDDs to Resilient Distributed Property Graphs
   b. Join Vertex Table and Edge Table to capture the relationships
   c. It is better in batch processing, it may not have good performance as compared to those graph specialised networks
3. **Neo4j**
   a. Graph database + Graph Processing
   b. SQL like interface: Cypher Query Language

Delta Lakes:

Databases:

- Designed to store structured data (i.e. table)
- Can be read through SQL queries
- Data adhere to **strict schema**
    - Allows database management system to heavily co-optimise data storage and processing through an optimized query processing engine
- Very **fast computation** and **strong transactional ACID** guarantees on **read/write operations**
- **Uses**: Online Transaction Processing (OLTP)

Data Warehouse:



- A central repository of integrated, historical data from multiple data sources

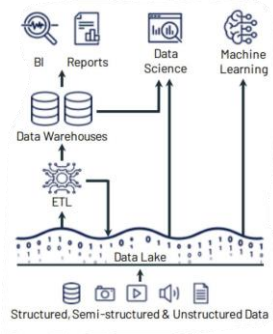
Benefits:

1. Served the business community well
    a. **Store large amounts** of historical data from different sources
    b. **Very reliable with strong transactional ACID** guarantees
    c. Modelled with standard start-schema modelling techniques
    d. Ideally suited for **business intelligence and reporting**

Challenges

1. **Big Data Trends** (Volume, Velocity, Variety, Veracity)
    a. Growth in data sizes
    b. Growth in the diversity of analytics
    c. Hard to address the big data trends
    d. **Extremely expensive to scale out** (true for most database systems)
    e. **Do not support non-SQL based** analytics very well.

Data Lake:



E.g. Hadoop MapReduce / Spark / NoSQL, Pregel

- **Cost effective** central repository to store data at any scale
- A **distributed storage solution**, runs on **commodity hardware**, and easily **scales out horizontally**
    - Distributed solution where some is focused on storage and some is focused on computation
- Data is **saved as files with open formats**
    - Any processing engine can read and write them using standard APIs

Types of Choices:

Note that if we want more specific tasks, we can make use of other engines. Else, a good starting point could be Spark since it is quite general

1. **Storage System**: HDFS, S3, Cloud and etc
2. **File Format**:
    a. Structured: Parquet, ORC
    b. Semi-Structured: JSON
    c. Unstructured formats: text, images, audio, video
3. **Computing/Processing engine(s)**:
    a. Batch processing engine: Spark, Presto, Apache Hive
    b. Stream processing engine: Spark, Apache, Flink
    c. Machine learning library: Spark MLlib, scikit-learn, R
    d. Graph processing: Pregel, Giraph

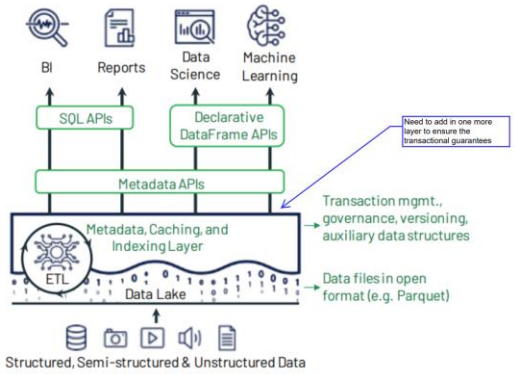
Benefits:

1. Helps to decouple the distributed storage system from the distributed compute system
    a. This allows for each system to scale out as needed by the workloads
    b. The company can choose which kind of storage system they want and also the format etc. Computing system can also be chosen
2. A much cheaper solution than databases → Explosive growth of the big data ecosystem

Cons:

1. Fails to provide ACID guarantees (Sacrifices the reliability)
2. Building and maintaining an effective data lake requires expert skills
3. Easy to ingest data but very expensive to transform data and make it meaningful to deliver business values
4. Data quality issues due to the lack of schema enforcement

Data Lakehouse:



- A system that merges both data lake and warehouse
  - The **flexibility, low cost**, and scale of a data lake
  - Data management and ACID transactions of data warehouses
- We just add in one more layer to ensure the transactional guarantees

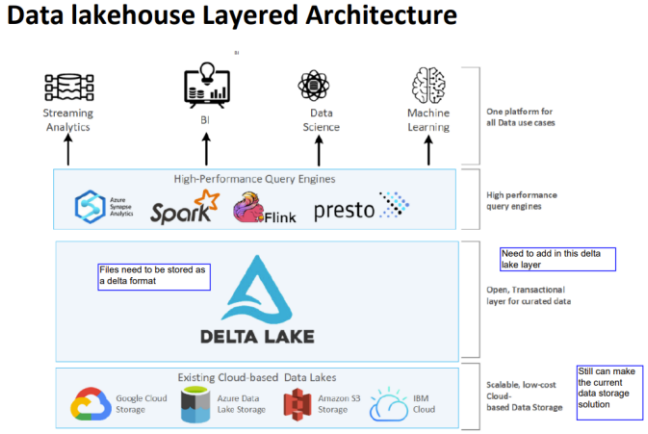
Type of users that would benefit:

1. Business Intelligence
2. Machine Learning/AI

Cloud Computing compatibility:

- With separate storage and computing resources

Delta Lake:



- The **metadata, caching and indexing layer** on top of a data lake storage that provides an abstraction layer to serve ACID transactions and other management feature.
- This is a **storage solution for the data lakehouse** additional layer that deals with the transactional guarantees

Benefits:

1. Transactional **ACID guarantees**
2. Full DML (Data Manipulation Language) support
3. Audit History
4. Unification of batch and streaming into one processing model
5. Schema enforcement and evolution
6. Rich metadata support and scaling

Format

- **Standard Parquet file** with no additional metadata
  - Parquet Files are column oriented: Performs compress on a column-by-column basis
  - It is also open source
  - Self-describing: Actual data + metadata (schema & file structure)
  - It is auto partitioned into 8 partitions once we save it. Once we extract the file, we will have the partitioned data
- **Delta Logs**
  - We will save it as delta instead. The delta log records the transaction logs. As long as we change the file or add file, there will be a transaction log inside.
  - Transaction log is an ordered record of every transaction made against a Delta table since it was created.
  - Just have to look at this to verify if records are made
  - ***Main Goal***: 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. If an operation is not recorded in the transaction log, it never happened.
  - Allows for ***scalable metadata handling***
  - Allows for ***time travel***

Types of Operations

Action	Description
Add file	Adds a file
Remove file	Removes a file
Update Metadata	Update the table's metadata (e.g. changing the table or file's name, schema or partitioning). The first transaction log entry for a table or file will always contain an Update Metadata action with the schema, the partition columns and other information.
Set transaction	Records that a structured streaming job has committed a micro-batch with the given stream ID. For more information, see Chapter X: Streaming
Change Protocol	Enables new features by switching the Delta Lake transaction log to the newst software protocol
Commit Info	Contains information about the commit, which operation was made, from where, and at what time. Every transaction log entry will contain a Commit Info action.