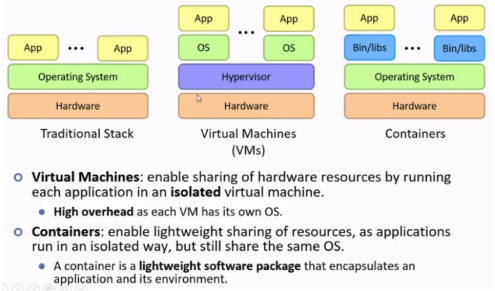
CS4225
AY24/25 SEM 1
github/Gavino3o

Introduction

- What is data science? The study of data. It involves developing methods of recording, storing, and analysing data to effectively extract useful information.
- Challenges of Big Data: Volume, Velocity, Variety, Veracity, Value.
- Cloud Computing: The delivery of computing services over the internet. Eg. storage, databases, networking, software, analytics, intelligence, etc.
- Virtualisation and Containers:



- Infrastructure as a Service (IaaS): Provides virtualised computing resources over the internet. Eg. AWS, Azure, Google Cloud.
- Platform as a Service (PaaS): Provides a platform allowing customers to develop, run, and manage applications without the complexity of building and maintaining the infrastructure.
- Software as a Service (SaaS): Software that is centrally hosted and licensed on a subscription basis.

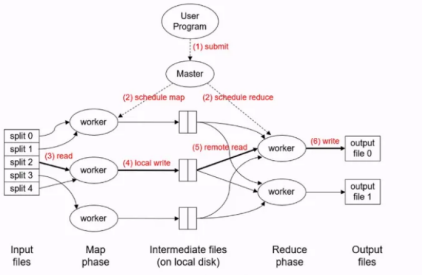
Data Centers

- A facility composed of networked computers and storage that businesses or other organisations use to organise, process, store, and disseminate large amounts of data.
- Single Server: A single physical server that runs a single operating system.
- Rack: Contains multiple servers, oftern connected to a rack switch (or top-of-rack switch).
- Data Center: Contains multiple racks, often connected to a data center switch (or a core switch).
- Bandwidth: The maximum amount of data that can be transmitted per unit time (e.g 1 Gbps). [Large Files]
- Latency: The time it takes for data to travel from the source to the destination (one-way) or from source to destination and back to source (round trip). Measured in ms. [Small Files]
- Throughput: Rate at which data is *actually* transmitted accross the network during a period of time.
- Latency combines additatively. The lower the latency, the better the performance.
- Bandwidth is approximately the minimum bandwidth throughout the network path. The higher the bandwidth, the better the performance.
- Storage Hierarchy (Higher the storage capacity, the higher the network latency, lower the bandwidth as it is bottlenecked by switches)
- Data Flow Paths (Local Disk -¿ Local DRAM -¿ TOR Switch -¿ Core Switch ... etc)
- Disk read/write is slower than DRAM read/write. DRAM read/write is slower than L1/L2 Cache read/write.
- Latency increases over the storage hiearchy as we go from Local Server to Rack to Data Center.
- **Latency is dominated by disk speed. Disk transfer is very expensive, any transfer involving disks take about the same amount of time.**
- Bandwidth decreases over the storage hiearchy as we go from Local to Rack to Dat Center
- **Bandwidth tends to be bottlenecked by networking switches.**
- Worst Performance: Disk read/write for Latency. Network switch bottleneck for Bandwidth.
- Ideas for Massive Data Processing:
  - Horizontal Scaling: Distributing the load accross multiple machines. Eg. Hadoop, Spark. Instead of vertical scaling (upgrading a single machine).
  - Move processing to the data
  - Process data sequentially and avoid random access as it involves disk seeks which leads to high latency.
  - Seamless scalability: The ability to increase the capacity of the system without affecting the performance of the system.

MapReduce (Hadoop)

- MapReduce is a programming model for processing and generating large data sets with a parallel, distributed algorithm on a cluster.
- Divide, Conquer, Aggregate.
- Challenges: Allocating Tasks, Scheduling, Partial Results, Aggregation, Synchronisation, Fault Tolerance.
- Synchronisation Barrier between Map phase and Reduce phase.
- Map Task is a basic unit of work. It is typically 128MB of data. At the beginning the input is broken into splits of 128MB. A map task is a job requiring to process one split, not a worker.
- Map Function is a user-defined function that processes a key/value pair to generate a set of intermediate key/value pairs.
- A map task can invoke the map function multiple times.
- Reduce Task is a basic unit of work. It is typically 1GB of data. A reduce task is a job requiring to process one partition, not a worker.
- Reduce Function is a user-defined function that processes a key and a set of values to generate a set of output values.
- A reduce task can invoke the reduce function multiple times.
- Data is **sorted by key and not value.**
- All values with the same key are sent to the same reduce task.
- Formally:
  - **Map**: $(k_1, v_1) \rightarrow list(k_2, v_2)$
  - **Reduce**: $(k_2, list(v_2)) \rightarrow list(k_3, v_3)$
- MapReduce execution framework handles scheduling, shuffling, synchronisaton, faults. Everything happens on top of a distributed file system, HDFS.
- MapReduce execution steps:
  Submit Job $\rightarrow$ Schedule Tasks $\rightarrow$ Read $\rightarrow$ MAP $\rightarrow$ Shuffle [Local Write, Remote Read] $\rightarrow$ REDUCE $\rightarrow$ Write.
- **Shuffle**: The process of transferring data from the map tasks to the reduce tasks. It involves sorting, partitioning, and copying intermediate data.
- **Partitioning (Local Write)**: The process of determining which reduce task will receive the intermediate data for a given key. Partitions are based of keys, a single partition can have multiple keys.
- **Sorting**: The process of sorting the intermediate data by key. Key value pairs arrive to the reduce task in order sorted by key.
- **Copying (Remote Read)**: The process of transferring the intermediate data from the map tasks to the reduce tasks. Reduce worker reads data from the **same corresponding partition from each Map worker**.
- Reduce phase is often the slowest phase as it involves network I/O and synchronisaton.
- The **Master Node** is responsible for scheduling tasks, monitoring tasks, and re-executing failed tasks. It does not execute tasks.
- The **Worker Node** is responsible for executing tasks. It does not schedule tasks. A single worker can handle multiple map tasks.

MapReduce Implementation



Q: What disadvantages are there if the size of each split (or chunk) is too big or small?
A: Too big: limited parallelism. Too small: high overhead (master node may be overwhelmed by scheduling work)

- The number of:
  - Map Function Calls = Number of Input Key-Value pairs
  - Reduce Function Calls = Number of Unique Intermediate Keys
  - Map Tasks = Number of Input Splits
  - Reduce Tasks = Number of Partitions (Specified by user)

Partitioner and Combiner

- Task Straggler Problem: Some tasks take longer to complete than others. This can slow down the entire job.
- It can be caused by certain keys in the data set that have too high of a frequency compared to other keys.
- User can define a custom partitioner to partition the data in a way that reduces the load on the reduce tasks.
- Combiner is a mini-reduce function that runs on the map side. It is used to reduce the amount of data that needs to be transferred to the reduce tasks.
- Generally, ensure that the computation is associative and commutative to ensure that the correctness of the combiner.
- Combiners are invoked after the map function, during the local write phase (but before the actual disk write to save disk I/O).
- Paritioners are invoked during the local write phase, as this stage requires knowing which keys need to go to which reducer.
- Both paritioners and combiners are optional and can be defined by the user.
- Guideline for basic algorithmic design: Linear scalibility, minimise disk and network I/O, reduce memory working set of each task/worker.

Secondary Sort

- In the Reduce phase, the intermediate data is sorted by key. However, we may want to sort the values for each key.
- Secondary Sort is the process of sorting the values for each key in the Reduce phase.
- Composite Key (k_1, k_2, v) + Custom Comparator (Compare k_1; If equal, compare k_2) + Custom Partitioner (Partition by k_1 only)
- It is not suitable to used hash based partitioning as it will not guarantee that all values for a key will go to the same reducer.
- Since we defined the custom comparator to sort based on the composite key, the values for each key will be sorted.
- This ultises the fact that the intermediate data is sorted by key in the shuffle phase.
- Sorting data in the reducer is expensive as it involves disk I/O. Sorting data in the map phase is cheaper as it involves memory I/O.

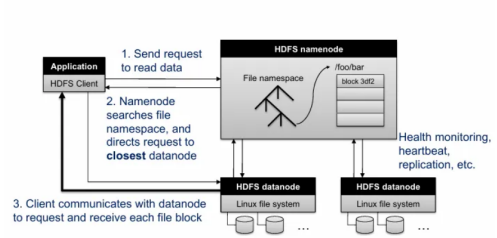
Preserving State in MapReduce

- In-mapper combiner: A combiner that runs within the map task. It is used to reduce the amount of data that needs to be transferred to the reduce tasks.
- Different calls to the function used in the mapping or reducing phase can preserve a state (e.g store variables) to be shared between the function calls.
- A key thing to note is that one of our goals is to reduce the amount of disk and memory IOs to maximise efficiency. However, with the context of the program, we will emit many intermediate results which incurs an IO each time.
- Word Count Example:
  - Version 1: Using a hash table in the mapping function. In a way, the mapping is already doing combining job using the properties of a hash table. [IN-MAPPER COMBINER as opposed to regular combiner (externally defined)]
  - For version 1, regular combiners are still used. As regular combiners can combine outputs for the whole map task, not just the map function. (I.e it can combine each hash table produced by each map function call)
  - Version 2: A single hash table is stored for the execution of the map task in the Mapper object. Each map function call will update the shared hash table in the Mapper object.
  - For version 2, a drawback is that the hash table is stored in memory (increased memory working set). If the hash table is too large, it may cause memory issues.

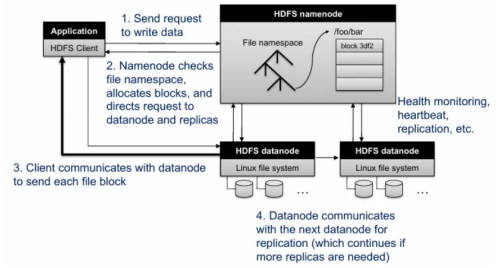
Hadoop Distributed File System (HDFS)

- Conceptually similar to a regular file system, but with the data spread accross a cluster of machines.
- Same data is replicated across multiple machines to ensure fault tolerance. Default replication factor is 3.
- Single master node (NameNode) and multiple worker nodes (DataNodes). Centralised metadata management.
- File stored as blocks (default 128MB). Blocks are replicated across multiple DataNodes.
- Write Once, Read Many (WORM) model. Once a file is written, it cannot be modified. But files can be appended to.
- HDFS is optimised for large files. It is not suitable for small files as it incurs overhead for each file.
- HDFS is designed for high throughput, not low latency. It is not suitable for real-time or random access.

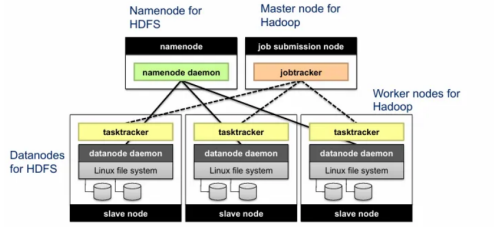
HDFS Architecture: Reading Data



HDFS Architecture: Writing Data



- NameNode: Manages the fiile system namespace (file/directory structure, metadata, file-to-block mapping, access permissions). Directs clients to datanodes. It does not store the data itself.
- NameNode also maintains overall health of the cluster through heartbeats, block rebalancing.
- Data Locality:



The same set of nodes are used for both HDFS and Hadoop. Hadoop tries to schedule map tasks to run on the machines that already contain the needed data (called **data locality**; or "moving the task to the data")

Relational Databases

- Projection (SELECT): Map-only job. Output of the map function is the final output desired.
- Selection (WHERE): Map-only job. Predicate is applied in the map function.
- Aggregation (GROUP BY, COUNT, SUM, AVG): Map-Reduce job. Map function emits intermediate key-value pairs. Reduce function aggregates the values for each key.
- Join (INNER JOIN, OUTER JOIN):
  - **Method 1: Braodcast Join (Map Join):**
  - Small table is broadcasted to all worker nodes. Large table is partitioned and sent to worker nodes.
  - The small table can be converted to a hash table and stored in memory.
  - Mapper iterates through the large table and looks up the corresponding value in the hash table for the join.
  - **Method 2: Common Join (Reduce-Side Join):**
  - Slower than broadcast join as it involves network I/O, but it uses less memory.
  - Different mapper tasks are used for each table. The reducer task is used to join the tables.
  - The key of intermediate results is appended with `table_id` to create a composite key. Secondary sort is used to ensure that keys from one table all arrive before keys from the other table.
  - The reducer task will hold the intermediate results from a table in memory, then iterate through the values from the other table and perform the join.

Similarity Search

- Near Neighbours: Objects that are similar to each other.
- Distance Measure: A function that measures the distance between two objects.
- Similarity Measure: A function that measures the similarity between two objects. Opposite of distance measure.
- Euclidean Distance: $\sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$
- Manhattan Distance: $\sum_{i=1}^{n}|x_i - y_i|$
- Cosine Similarity: $\frac{A \cdot B}{||A|| \cdot ||B||}$ where $A \cdot B = \sum_{i=1}^{n} A_i \cdot B_i$
- Cosine Similarity is often used for text data as it is invariant to the magnitude of the vectors. Only considers the direction of the vectors.
- Jaccard Similarity: $\frac{|A \cap B|}{|A \cup B|}$. Intersection over Union.
- Jaccard Distance = 1 - Jaccard Similarity
- **All pairs similarity search**: Given a large number of documents, find pairs of documents that are similar to each other.
- **Similarity search:** Given a query document, find the similar documents in the database.
- **Shingling:** A technique to convert a document into a set of shingles. A shingle is a sequence of words in a document. Imagine sliding window of size k over the document.
- **Min-Hashing:** A technique to estimate the Jaccard similarity between two sets. It is used to reduce the size of the shingle set.

Shingles

- Matrix Representation: Each row represents a shingle, each column represents a document.
- A cell is 1 if the shingle is in the document, 0 otherwise.
- Intersection: Number of rows with both columns as 1.
- Union: Number of rows with at least one column as 1.
- K-Shingle Set: Set of all possible k-shingles in a document.

Min-Hashing

- **Key Property:** The probability that two documents have the same min-hash value is equal to the Jaccard similarity of the documents.
- Formally: $P(h(A) = h(B)) = J(A, B)$
- **Key Idea:** Use a hash function to map the shingle set to a smaller set of hash values. The minimum hash value is used as the signature of the document.
- Multiple hash functions are used to generate multiple signatures for each document. The signatures are used to estimate the Jaccard similarity.
- Documents with Jaccard similarity above a threshold will be accepted as candidate pairs.
- Candidate pairs can themselves be the output or can be further verified using the original shingle set.
- **MapReduce Framework:**
  - Map: Read over the document and extract the shingles. Hash each shingle and take the min of them to get the MinHash signature
  - Emit <`signature, document_id`>. *Notice that the signature is the key*
  - Redicue: Receive all documents with a given MinHash signature. Generate all candidate pairs from these documents.
  - (Optional): Verify the candidate pairs using the original shingle set.

K-Means Clustering

- K-Means is an iterative algorithm that partitions the data into K clusters.
  1. Randomly initialise K cluster centroids.
  2. Assign each data point to the nearest cluster centroid.
  3. Recompute the cluster centroids based on the data points assigned to the cluster.
  4. Stop when the cluster centroids do not change significantly or no data points change clusters.
- Map Phase: Assign each data point to the nearest cluster centroid. Emit <`cluster_id, point`>.
- Reduce Phase: Recompute the cluster centroids based on the data points assigned to the cluster. Emit <`cluster_id, new_centroid`>.
- Extending Point: Adding a dimension to the coordinate emitted which is just a 1 to help with the computation of the sum and count.
- V1: Emits each point with the cluster it belongs to without combining.
- Disk I/O exchanged: $O(nmd)$. Where n is the number of data points, m is the number of iterations, d is the number of dimensions.
- V2: In-mapper combiner combines points in the same cluster within the map task using a hash table. Emits each cluster and the list of points assigned to it in the current iteration.
- Disk I/O exchanged: $O(kmd)$. Where k is the number of clusters.

MapReduce Implementation v2 (with in-mapper combiner)

```
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)
```

This MapReduce job is similar to v1, but uses an in-mapper combiner to combine points in the same map task using a data structure H.

NoSQL Databases

- NoSQL databases are non-relational databases that provide a mechanism for storage and retrieval of data that is modelled in means other than the tabular relations used in relational databases.
- Stands for "Not Only SQL"
- Examples: Key-Value Stores, Document Stores, Wide Column Stores, Graph Databases, Vector Databases.
- Features: Horizontally Scalable, Replicated/Distributed over many servers, Schema-less, Eventual Consistency, High Availability, Simple call interface, often weaker concurrency model than RDBMS, Effcient use of distributed indexes and RAM, Flexible Schemas.
- Pros: Flexible/Dynamic Schema, Horizontal Scalability, High Performance and Availability
- Cons: No declarative query language, Weaker consistency guarantees.

Key-Value Stores

- **Keys** are usually primitive data types (ints, string, raw bytes, etc) and can be queried.
- **Values** can be any data type (string, JSON, XML, BLOB, etc) and are opaque to the database. It usually does not support querying.
- **Operations**: Get, Put,Multi-Get, Multi-Put, Range Queries.
- Suitable for:
  - Small continuous reads and writes
  - Storing basic information (user profiles, session information, caches, raw chunks of bytes). Or no clear schema.
  - When compex queries are not needed.
  - Non-persistent: Big in-memory hash table. Example: Memcached, Redis.
  - Persistent: Data is stored persistently on disk. Example: RocksDB, Dynamo, Riak

Document Stores

- Basically Hierarchical Key-Value Stores. Values are documents (JSON, XML, etc).
- Collections = Tables, Documents = Rows, Fields = Columns.
- Unlike Key-Value Stores, the database can query the values.
- Example Document Stores: **MongoDB**, CouchDB, Couchbase.

- The $ operator represents a special key in MongoDB. It is used to project elements in an array.
- 1 stands for true/include, 0 stands for false/exclude in MongoDB.

CRUD: Update

In MongoDB

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

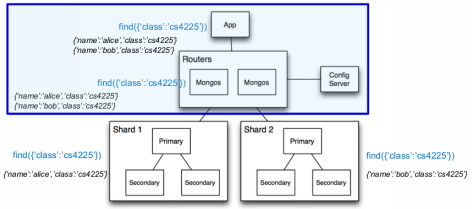
In SQL

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

MongoDB

- Routers(mongos): Handles requests from application and route the queries to the correct shards.
- Config Server: Stores metadata and configuration settings for the cluster (e.g. which data is in which shard)
- Shards: Data is partitioned based on a shard key(variable of the record). Within each shard, data is stored in replica sets for redundancy and fault tolerance.
- Sharding allows for horizontal scalability.

Example of Read or Write Query



1. Query is issued to a router (`mongos`) instance
2. With help of config server, `mongos` determines which shards to query
3. Query is sent to relevant shards (*partition pruning*)
4. Shards run query on their data, and send results back to `mongos`
5. `mongos` merges the query results and returns the merged results to the application

- Partition Pruning: Only the relevant shards are queried. The query is sent to the relevant shards based on the shard key.
- If query is not based on the shard key, the query is sent to all shards. This is called scatter-gather.
- Supports join queries through denormalisation. Data is duplicated across multiple collections to avoid joins.

Replication in MongoDB

- Standard Configuration: 3 nodes, 1 primary, 2 secondaries.
- Primary receives all write operations.
- Secondaries replicate the primary's **operation log** and apply the operations.
- Users can configure the read preference to allow reading from secondaries.
- Secondaries have eventual consistency. They may lag behind the primary. But allowing reads from secondaries can improve decrease latency and distribute load (improving thoughput).

Wide Column Stores

- Column Families: A collection of columns. Each column family has a unique name.
- Sparsity: Columns can be added to a row without adding them to all rows.
- If a column is not used for a row, it does not use any space.
- Example: BigTable, HBase, Cassandra.

Vector Databases

- Each row is a vector. Each column is a dimension.
- Usually used to store dense, numerical and high-dimensional data.
- Allows fast similarity search and clustering.
- Major Types of algorithm: Min-Hashing, Locality Sensitive Hashing (LSH).
- Features: Scalability, Real-Time Updates, Replication, Fault Tolerance, High Availability.
- Examples: Milvus, Redis, Weaviate, MongoDB Atlas.
- Commonly used for AI/ML applications, especially large language models are often used to convert text into vectors which are used for search, recommendation, clustering.
- Similarly, vision models convert images into embeddings.

Strong Vs Eventual 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 given enough time, eventually all reads will return the last written value.
- RDBMS provides ACID properties (Atomicity, Consistency, Isolation, Durability).
- NoSQL databases often provide BASE properties (Basically Available, Soft State, Eventually Consistent).
  - Basically Available: Basic reading and writing operations are available most of the time
  - Soft State: Without guarantees, we only have some probability of knowing the state at any time.
  - Eventually Consistent: The system will eventually become consistent.
- Eventual consistency offers better availability at the cost of weaker consistency guarantee. It is suitable for applications that can tolerate stale data.

Distributed Databases

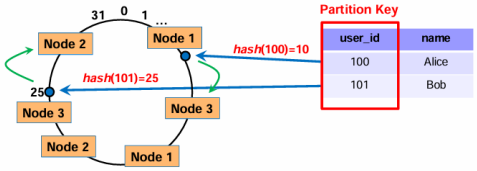
- Advantages: Scalability, Availability, Fault Tolerance, Lower Latency (Nearest Replica)
- Architectures: Shared Everything, Shared Memory, Shared Disk, Shared Nothing.
- RDBMS and NoSQL databases often use the Shared Nothing architecture.
- Network conditions between nodes may fail. In some cases, the network may be partitioned.
- **CAP Theorem:** Consistency, Availability, Partition Tolerance. A distributed system can only guarantee two of the three.
  - Consistency: Every read receives the most recent write or an error. (Strong Consistency)
  - Availability: Every request receives a response.
  - Partition Tolerance: The system continues to operate despite network partitions.
- Intutition: In the event of a network partition, the system can either choose to be consistent or available.
- Generally, NoSQL databases choose to be available and partition tolerant (AP). RDBMS choose to be consistent and partition tolerant (CP).
- But databases often have tunable consistency levels.

Data Partitioning

- **Table Partitioning:** Put different tables (or collections) on different nodes.
- Problem: Scalability is an issue as each table cannnot be split accross multiple nodes.
- **Horizontal Partitioning (Sharding):** Different rows (or documents) of the same table are stored on different nodes.
- Records are partitioned based on a shard key. The shard key is used to determine which node the record is stored on.
- We should choose a shard key based on the access patterns of the data (e.g. groupBy). It should be evenly distributed to avoid hotspots.
- We want to ultilise partition pruning to reduce the number of nodes that need to be queried.
- **Low Cardinality:** A shard key with low amounts of unique values. It is not suitable for sharding as it will lead to hotspots.
- **High Frequency:** A single shard key with high amounts of records. It is not suitable for sharding as it will lead to hotspots.
- To mitigate these issues, we can use *composite shard keys*.
- **Range Partitioning:** We can combine horizontal paritioning with range partitioning. Each node is responsible for a range of shard keys.
- Beneficial for range queries as the query can be sent to the relevant node.
- **Hash Partitioning:** Each node is responsible for a range of hash values of the shard key.
- It can still lead to hotspots if the hash function is not well distributed.

Consistent Hashing

- Weakness of Hash/Range Partitioning: How do we distribute the data when we add or remove nodes?
- If we redo the partitioning, we will have to move a lot of data which is inefficient.
- **Consistent Hashing:** A technique that minimises the amount of data that needs to be moved when nodes are added or removed.
- Think of the output of hashing the shard key as on a circle.
- Each node has a marker on the circle. Each node can even have multiple markers.
- Each record is assigned to the node in the clockwise direction of the first marker it encounters.
- To delete a node, we simply remove the marker from the circle. The records that were assigned to the node will be assigned to the next node in the clockwise direction.
- To add a node, we simply add a marker to the circle. The records that were assigned to the next node in the clockwise direction will be assigned to the new node.
- Replication Strategy: Each record is assigned to the next n nodes in the clockwise direction.
- Benefits of multiple markers: Load balancing, when we remove a node, the records are distributed to different nodes in their clockwise direction.

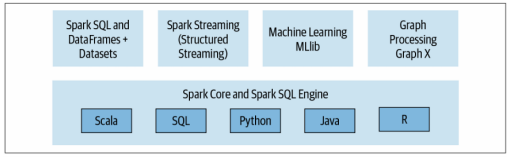


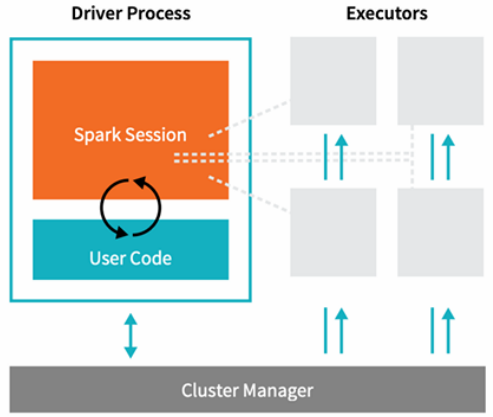
Apache Spark

Hadoop vs Spark

- **Hadoop**: Disk-based, MapReduce, HDFS. Not suitable for iterative algorithms as it incurs network and disk I/O overhead for intermediate data.
- **Spark**: In-memory, DAG, RDDs. In the event memory is insufficient, Spark spills data from memory to disk.

Spark Architecture and APIs





- **Driver Process**: Manages the execution of the Spark job. Responds to user inputs. Distribute work to the executors.
- **Cluster Manager**: Manages the resources of the cluster. Eg. YARN, Mesos, Kubernetes.
- **Worker Node**: Runs the executors.
- **Executor**: Runs tasks and keeps data in memory or disk storage across them.
- **RDDs**: Resilient Distributed Datasets. A collection of JVM objects. Functional operators (map, filter, etc) are applied to RDDs to transform them.
- **DataFrames**: A distributed collection of data organized into named columns. Similar to a table in a relational database. Expression-based transformation operations. Logical plans and optimisers.

- **Datasets**: A distributed collection of data with a known schema. Combines the benefits of RDDs and DataFrames. Internally rows, externally JVM objects. Typs safe and fast.

RDDs

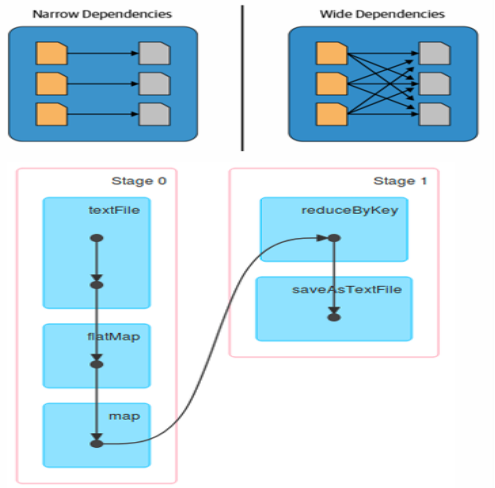
- **Resilient Distributed Datasets**
- Fault-tolerant collection of elements that can be operated on in parallel
- Immutable, partitioned collection of records
- Distributed across a cluster of machines.

Transformations and Actions

- Transformations are operations that create a new RDD from an existing one.
- Lazy evaluation: Transformations are not executed immediately. They are only executed when an action is called. This allows Spark to optimise the execution plan.
- Example: `map`, `filter`, `flatMap`, `groupByKey`, `reduceByKey`, `join`, `order`, `select`
- Actions are operations that return a value to the driver program after running a computation on the dataset.
- Example: `collect`, `count`, `reduce`, `saveAsTextFile`, `foreach`, `take`, `show`
- Execution of transformations and actions are executed in parallel accross different worker machines as RDDs are distributed across different worker machines. Results are returned to the driver program in the final step.
- Caching: Persisting RDDs in memory across operations. Useful for iterative algorithms.
- `cache()` is a transformation that persists the RDD in memory.
- `persist(options)` is an action that allows for more control over the persistence of the RDD.
- `unpersist()` is an action that removes the RDD from memory.
- We should cache RDDs that are used multiple times in the computation or when it is expensive to recompute the RDD.
- If we did not cache the RDD, Spark will recompute the RDD each time it is used in an action.
- When worker nodes have insufficient memory, Spark may evict LRU RDDs from memory to disk.

Directed Acyclic Graph (DAG)

- A DAG is a graph with directed edges and no cycles.
- In Spark, the DAG is a logical representation of the computation.
- Transformations construct the DAG; actions execute the DAG.
- Spark optimises the DAG by combining operations and minimising data shuffling.
- Narrow dependencies: Each partition of the parent RDD is used by at most one partition of the child RDD. Example: `map`, `filter`, `flatMap`, `contains`
- Wide dependencies: Each partition of the parent RDD may be used by multiple partitions of the child RDD. Example: `groupByKey`, `join`, `reduceByKey`, `sortByKey`, `orderByKey`
- In the DAG, consecutive narrow transformations are combined into a single stage and executed on the same machines. Wide transformations are separated into different stages.
- Across stages, data is shuffled across the network, which involves writing intermediate data to disk.
- Spark tries to minimise the number of stages and the amount of data shuffled.
- **Lineage**: The sequence of transformations that lead to an RDD.
- If a worker node fails, Spark can recompute the lost partitions of an RDD using the lineage. Note: we only need to recompute the lost partitions, not the entire RDD.



DataFrames

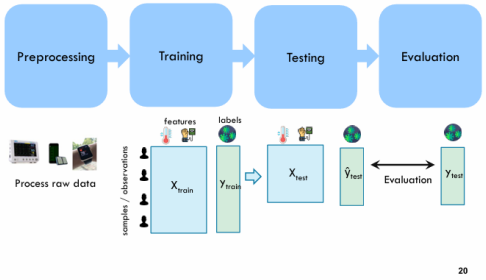
- A distributed collection of data organised into named columns.
- Similar to a table in a relational database.
- Easier to use than RDDs as it has a higher-level API.
- All Dataframe operations are still ultimately compiled down to RDD operation by Spark.
- Generally, transformation functions take in either strings or column objects.
- Transformations are still lazyly evaluated.

Spark SQL

- Spark SQL is a module for working with structured data.
- It provides a programming abstraction called DataFrames and can also act as a distributed SQL query engine.
- Spark SQL provides a domain-specific language for working with structured data.
- It allows running SQL queries on existing RDDs and DataFrames.
- Catalyst optimiser is the Spark SQL query optimiser. It takes a computational query and converts it into an optimised logical plan. Four Phases: Analysis, Logical Optimisation, Physical Planning, Code Generation (Project Tungsten).
- Multiple physical plans can be generated for a single logical plan. The optimiser chooses the best physical plan based on cost estimation.
- Project Tungsten is the Spark SQL execution engine. It aims to improve performance by optimising memory usage and CPU utilisation.
- Tungsten optimises memory usage by using binary processing, cache-aware computation, and code generation.
- **Unified API**: Spark SQL can be used with Java, Scala, Python, SQL, and R. It has one engine for all types of data processing.

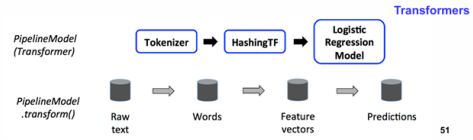
Machine Learning with Spark

Typical Machine Learning Pipeline



- Data Quality: Missing values (impute, drop, add column indicating it is missing or not)
- Categorical Encoding: Convert categorical variables to numerical variables. Numerical values are often assigned in a way that represents the ordinal relationship between the categories or inherent order among the categories.
- One Hot Encoding: Convert categorical variables to binary vectors. Each category is represented by a binary vector. Useful when there is no ordinal relationship between the categories, and to ensure that the categorical variable does not imply any numerical relationship.
- Normalization: Scale the features to a standard range. Useful for algorithms that are sensitive to the scale of the input features. Example: clipping, log transform, standard scalerm, min-max scaler.
- Logistic Regression: A linear model for binary classification. It models the probability that the output is 1 given the input features. Ultilses the sigmoid function. $\sigma(x) = \frac{1}{1+e^{-x}}$
- $\hat{y} = \sigma(x \cdot w + b)$
- Cross Entroypy Loss: Measures the difference between two probability distributions. It is used as the loss function for logistic regression.
  $L(y, \hat{y}) = -y \log(\hat{y}) - (1-y) \log(1-\hat{y})$
- Gradient Descent: An optimisation algorithm that minimises the loss function. It iteratively updates the weights and biases in the direction of the negative gradient of the loss function.
- $w_{t+1} = w_t - \alpha \nabla_w L(w_t)$
- Evaluation Metrics: Accuracy, Precision, Recall, F1 Score, ROC Curve, AUC.
- Accuracy: $\frac{TP+TN}{TP+TN+FP+FN}$
- Precision: $\frac{TP}{TP+FP}$
- Recall: $\frac{TP}{TP+FN}$
- F1 Score: $2 \times \frac{Precision \times Recall}{Precision+Recall}$
- Errors: Mean Squared Error, Mean Absolute Error, Root Mean Squared Error.
- Mean Squared Error: $\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y_i})^2$
- Mean Absolute Error: $\frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y_i}|$
- Root Mean Squared Error: $\sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y_i})^2}$
- R Squared Value (0 to 1): Measures the proportion of the variance in the dependent variable that is predictable from the independent variable. The higher the better.

Pipelines



- Benefits: Better code reuse, Easier to perform cross validation, Easier to tune hyperparameters, Easier to productionise the model.
- Transformers are the building blocks of a pipeline.
- A transformer has a transform() method that takes in a DataFrame and returns a new DataFrame.
- Example: VectorAssembler, StringIndexer, OneHotEncoder, StandardScaler, LogisticRegression
- Generally, these transformers output a new DataFrame which append their result to the original DataFrame.
- A fitted model (e.g LogisticRegressionModel) is also a transformer. It transforms Dataframe by adding a prediction column.
- Estimators is an algorithm that takes in data, and outputs a fitted model. Example: A learnign algorithm like LogisticRegression can be fit to data, producing the trained logistic regression model.
- Estimators have a `fit()` method that takes in a DataFrame and returns a Transformer.

```
from pyspark.ml.classification import LogisticRegression

training = spark.read.format("libsvm").load("data/mllib/sample_libsvm_data.txt")

lr = LogisticRegression(maxIter=10)

lrModel = lr.fit(training)

print("Coefficients: " + str(lrModel.coefficients))
print("Intercept: " + str(lrModel.intercept))
```

- Pipelines are a sequence of stages. Each stage is either a Transformer or an Estimator.
- Pipeline itself is an Estimator. It has a `fit()` method that takes in a DataFrame and returns a PipelineModel.
- When `fit()` is called on a Pipeline, the stages are executed in order. For Transformers, the `transform()` method is called. For Estimators, the `fit()` method is called.
- The output of `Pipeline.fii()` is a PipelineModel, which is a Transformer, and consists of a series of Transformers.
- The `transform()` method of the PipelineModel applies the fitted model to the input DataFrame.

```
# Prepare training documents from a list of (id, text, label) tuples.
training = spark.createDataFrame([
    (0, "a b c d e spark", 1.0),
    (1, "b d", 0.0),
    (2, "spark f g h", 1.0),
    (3, "hadoop mapreduce", 0.0)
], ["id", "text", "label"])
```

```
# Configure an ML pipeline, which consists of three stages: tokenizer, hashingTF, and lr.
tokenizer = Tokenizer(inputCol="text", outputCol="words")
hashingTF = HashingTF(inputCol=tokenizer.getOutputCol(), outputCol="features")
lr = LogisticRegression(maxIter=10, regParam=0.001)
pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])
```

```
# Fit the pipeline to training documents.
model = pipeline.fit(training)
```

```
# Make predictions on train documents and print columns of interest.
pred_train = model.transform(training)
pred_train.drop('rawPrediction').show(truncate = False)
```

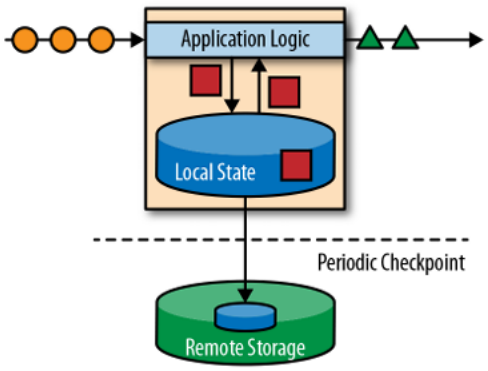
```
# Prepare test documents
test = spark.createDataFrame([
    (4, "spark i j k", 1.0),
    (5, "l m n", 0.0),
    (6, "spark hadoop spark", 1.0),
    (7, "apache hadoop", 0.0)
], ["id", "text", "label"])
```

```
# Make predictions on test documents and print columns of interest.
pred_test = model.transform(test)
pred_test.drop('rawPrediction').show(truncate = False)
```

```
# compute accuracy on the test set
predictionAndLabels = pred_test.select("prediction", "label")
evaluator = MulticlassClassificationEvaluator(metricName="accuracy")
print("Test set accuracy = " + str(evaluator.evaluate(predictionAndLabels)))
```

```
▸ (1) Spark Jobs
▸ predictionAndLabels:  pyspark.sql.dataframe.DataFrame = [prediction: double, label: double]
Test set accuracy = 0.75
```

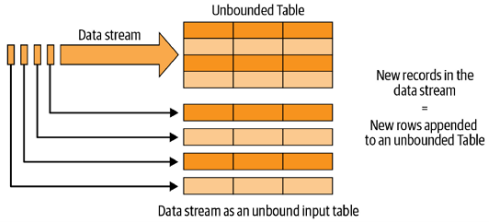
Stream Processing



Goal: Process data in real-time as it is generated. State can be stored and accessed in many different places including program variables, local files, or embedded/external databases.

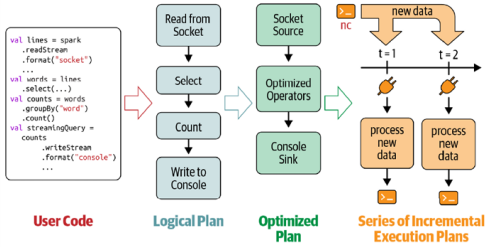
Micro-Batch Stream Processing

- Spark Structured Streaming uses a micro-batch processing model.
- Data from the input stream is divided into micro batches, each of which will be processed in the Spark cluster in a distributed manner.
- Small deterministic tasks generate the output of each micro-batch. Time is divided into small intervals, and data is processed in each interval.
- Advantages: quick; recover from failures efficently; deterministic nature ensures end-to-end exactly-once processing.
- Disadvantages: latency (cannot handle millisecond); micro-batch size affects latency and throughput; micro-batch processing can be less efficient than true stream processing.
- It is sufficient for most use cases.
- In Spark Structured Streaming, the input data is treated as an unbounded table.

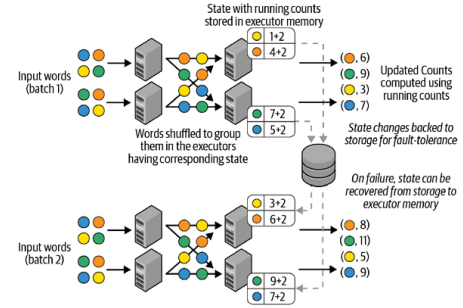


- Five Steps to Define Steaming Query:
  1. Define the input source.
  2. Transform Data.
  3. Define output sink and output mode.
  4. Specifying processing details. (Triggering details, checkpointing, etc)
  5. Start the query.

Incremental execution of streaming queries



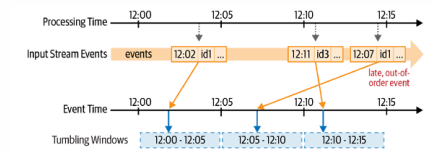
Distributed state management in Structured Streaming



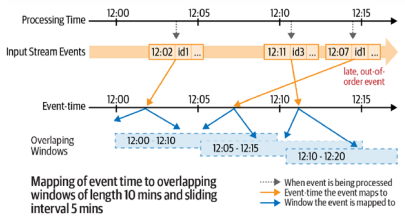
Data Transformation

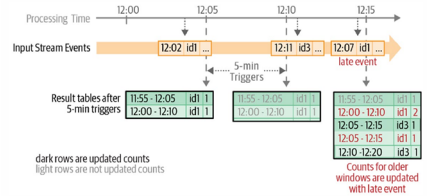
- **Stateless Transformations:**
  - Process each row individually without needing information from previous rows
  - Projection operations: `select()`, `explode`, `map()`, `flatMap()`
  - Selection operations: `filter()`, `where()`
- **Stateful Transformations:**
  - Process each row based on information from previous rows.
  - Example: `DataFrame.groupBy().count()`
  - In every micro-batch, the incremental plan adds the count of new records to the previous count generated by the previous micro-batch
  - The state is maintained in the memory of the Spark executors and is checkpointed to the configured location to tolerate failures.
- **Stateful Streaming Aggregations:**
  - **Aggregations not based on time windows:**
    - Global aggregations: `groupBy().count()`
    - Grouped aggregations: `groupBy("sensorId").mean("value")`
    - Supported aggregations: `count()`, `sum()`, `avg()`, `min()`, `max()`, `countDistinct()`, `collect_set()`, `approx_count_distinct()`
  - Processing Time: The time at which the data is processed by the system. (Not deterministic, susceptible to system delays)
  - Event Time: The time at which the event occurred in the real world. (Deterministic)
  - Event time decouples the processing speeed from the results. An event time window computation will yield the same result regardless of the processing time.
  - Watermark: A threshold that determines how late the data can be in event time. Data that arrives later than the watermark is considered late data.
  - **Aggregations with Event-Time Windows:**
    - Example:
      `sensorReadings.groupBy("sensorId", window("eventTime", windowLength, shiftAmt)).count()`

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



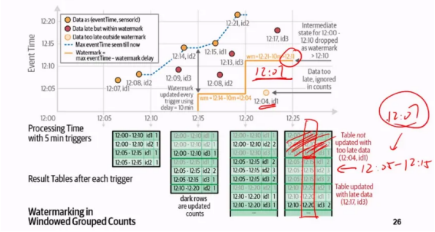

```
(sensorReadings
  .groupBy("sensorId", window("eventTime", "10 minute", "5 minute"))
  .count())
```





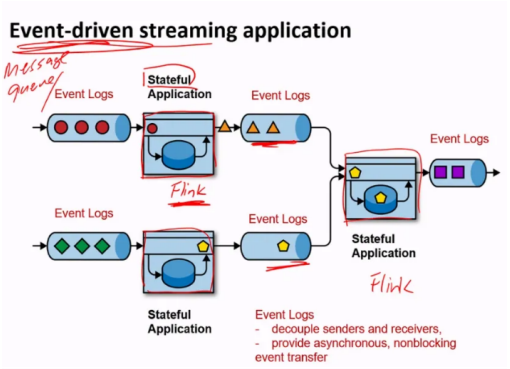
Handling Late Data with Watermarks

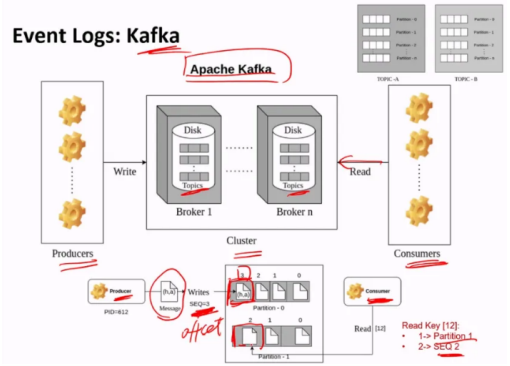
```
(sensorReadings
  .withWatermark("eventTime", "10 minutes")
  .groupBy("sensorId", window("eventTime", "10 minutes", "5 minutes"))
  .count())
```



- Take the latest **event time** and minus the watermark time. Let go of the rows in the result window with end interval time lower than calculated time.
- Watermark just determines which window records in the table to drop off. If it was not dropped, it may still be updated even though the late data event time is before the watermark time.
- Data with event time of 12:07 is still updated as the event window [12:05 — 12:15] is not dropped as the window end interval time is not before the watermark time.
- **Performance Tuning:**
  - Cluster resource provisioning appropriately to run 24/7
  - Number of partitions for shuffles to be set much lower than batch queries
  - Setting source rate limits for stability
  - Multiple streaming queries in the same Spark application
  - Tuning Spark SQL Engine

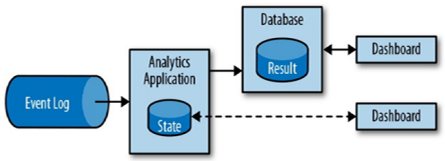
Flink





A Streaming Analytics Application

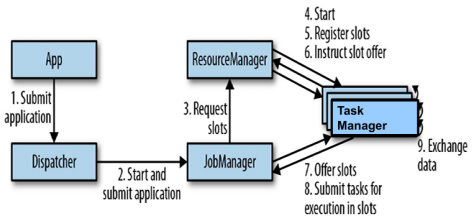
a stream processor takes care of all the processing steps, including event ingestion, continuous computation including state maintenance, and updating the results.



Much shorter time needed for an event to be incorporated into an analytics result !

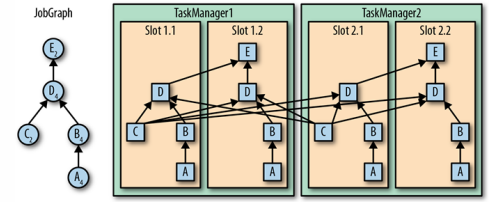
System Architecture

- Flink is a distributed system for stateful parallel data stream processing.

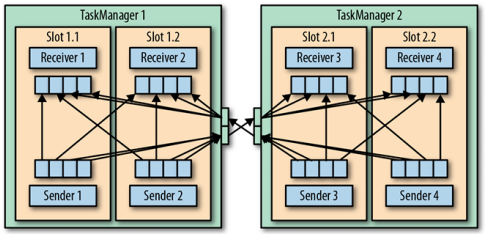


Task Execution

- A Task Manager can run multiple tasks concurrently
  - Tasks of the same operator (data parallelism)
  - Tasks of different operators (task parallelism)
  - Tasks of different applications (job parallelism)
- Task Managers offers certain number of processing slots to control the number of tasks it is able to concurrently execute. Think of slots as CPU cores.
- A processing slot can execute one slice of an application, i.e one parallel task of each operator of the application



- The tasks of a running application are continuously exchanging data.
- Task Managers take care of shipping data from sending tasks to receiving tasks.
- The network component of a Task Manager collects records in buffers before they are shipped.

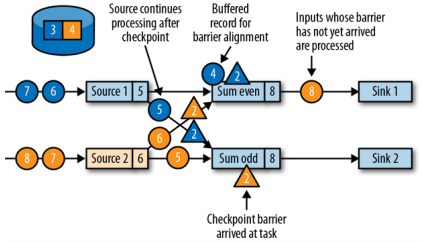


Event-Time Processing in Flink

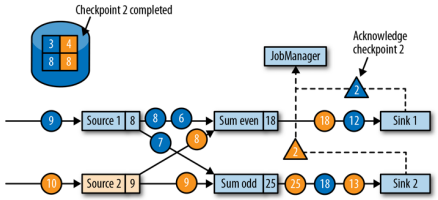
- Flink allows for event-time processing by allowing the user to assign **timestamps** to events/records.
- Flink can handle out-of-order events by using watermarks.
- **Watermarks** are used to derive the current event time at each task in an event-time application.
- In Flink, watermarks are implemented as special records holding a timestamp as a Long value. Wtermarks flow in a stream of regular records with annotated timestamps.
- Watermarks in Flink is not directly invovled in late data handling. It is used to determine when to emit results of windowed operations.

State Management

- Stateful Stream Processing Task: All data maintained by a task and used to compute the results of a function belong to the state of the task.
- **State Backend:**
  - Local State Management: A task of a stateful operator reads and updates its state for each incoming record.
  - Each parallel task locally maintains its state in memory to ensure fast state accesses.
- **Checkpointing:**
  - A Task Manager process may fail at any point, hence its storage must be considered volatile
  - Checkpointing the state of a task to a remote and persistent storage
  - The remote storage for checkpointing could be a distributed filesystem or a database system
  - *Consistent Checkpoints* (similar to Spark's micro-batch checkpointing, Stop the world, not used by Flink)
    1. Pause the ingestion of all input streams
    2. Wait for all in-flight data to be completely processed, meaning all tasks have processed all their input data
    3. Take a checkpoint by copying the state of each task to a remote, persistent storage. The checkpoint is complete when all tasks have finished their copies.
    4. Resume the ingestion of all streams.
  - To execute failure recovery from consistent checkpointing. Simply restart the application, reset the state of all staetful tasks to latest checkpoint, and resume tasks.
  - **Flink Checkpointing Algorithm**
    - Chandy-Lamport Algorithm: A distributed algorithm for recording a consistent global snapshot of a distributed system.
    - Does not pause the application but decouples checkpointing from processing
    - Some tasks contninue processing while others persist their state
    - Uses **checkpoint barrier**, a special record that signals the tasks to persist their state
    - Checkpoint barriers are injected by source operators into the regular stream of records and cannot overtake or be passed by other records
    - A checkpoint barrier carries a checkpoint ID to identify the checkpoint it belongs to and logically splits a stream into two parts
    - All state modifications due to records that precede a barrier are included in the barrier's checkpoint and all modifications due to records that follow the barrier are included in a later checkpoint.



- Job manager initiates checkpoints by sending message to all sources.
- Sources checkpoint their state and emit a checkpoint barrier.
- Records from input streams for which a barrier already arrived are buffered.
- All other records are regularly processed
- Tasks checkpoint their state once all barriers have been received, then they forward the checkpoint barrier
- Tasks continue regular processing after the checkpoint barrier is forwarded
- Sinks acknowledge the reception of a checkpoint barrier to the JobManager
- A checkpoint is complete when all tasks have acknowledged the successful checkpointing of their state

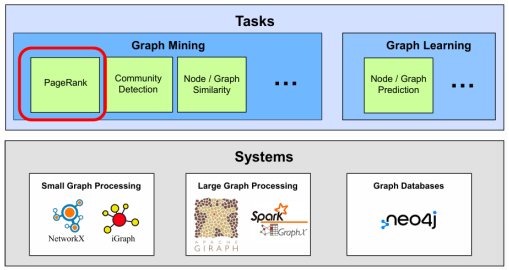


Conclusion: a comparison between Spark vs. Flink

- Spark
  - Microbatch streaming processing (latency of a few seconds)
  - Checkpoints are done for each microbatch in a synchronous manner ("stop the world")
  - Watermark: a configuration to determine when to drop the late events
- Flink
  - Real-time streaming processing (latency of milliseconds)
  - Checkpoints are done distributedly in an asynchronous manner (more efficient ➔ lower latency)
  - Watermark: a special record to determine when to trigger the even-time related results
    - Flink uses late handling functions (related to watermark) to determine when to drop the late events

Graphs

Graph Processing



Simple PageRank

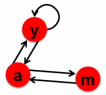
- We can visualise the web as a directed graph where each page is a node and each hyperlink is an edge.
- PageRank is an algorithm that measures the importance of a page in a network.
- The importance of a page is determined by the number of incoming links and the importance of the pages that link to it.
- If we assume incoming links are harder to manipulate, we can rank each page based on the number of incoming links.
- Problem: Malicious users can create a large number of pages that link to a target page to increase its rank.
- Solution: Make the rank of a page dependent on the rank of the pages that link to it.
- Therefore, links from important pages count more, this is true recursively.
- PageRank recursively computes the rank of a page based on the ranks of the pages that link to it. [Weighted edges]
- **Voting Formulation:**

- For each page j, we define its importance as r_j
- If page j with importance r_j has n outgoing links, each link gets $\frac{r_j}{n}$ importance.
- Page j's own importance is the sum of the votes on its incoming links.
- Formally: $r_j = \sum_{i \to j} \frac{r_i}{d_i}$.

  Where d_i is the number of outgoing links from page i.
- The sum of importances of pages linking to j, each divided by number of outgoing links from that page.
- In the event that the flow equations have no unique solution, we can add an additional constain to force uniqueness:
  $r_a + r_b + r_c = 1$

- **Matrix Formulation:**
  - Stochastic adjacency matrix, M
  - Let page i has d_i, out-links.
  - If $i \to j$, then $M_{ji} = \frac{1}{d_i}$, else $M_{ji} = 0$
  - M is a column-stochastic matrix, whjen each column sums to 1.
  - Let **Rank Vector** $= r$ and r_i is the rank of page i.
  - $\sum_i r_i = 1$
  - The flow equation can be written as $r = M \cdot r$
  - Column points to row. Each column is a page, each row is a page.

Example: Flow Equations & M



	y	a	m
y	½	½	0
a	½	0	1
m	0	½	0

$r = M \cdot r$

$r_y = r_y/2 + r_a/2$
$r_a = r_y/2 + r_m$
$r_m = r_a/2$

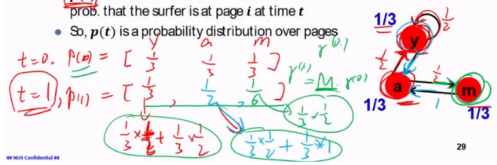
$$\begin{bmatrix} r_y \\ r_a \\ r_m \end{bmatrix} = \begin{bmatrix} ½ & ½ & 0 \\ ½ & 0 & 1 \\ 0 & ½ & 0 \end{bmatrix} \begin{bmatrix} r_y \\ r_a \\ r_m \end{bmatrix}$$

- **Power Iteration Method:**
  - Each node starts with equal importance (of $\frac{1}{N}$). During each step, each node passes its current importance along its outgoing edges, to its neighbors.
    1. Suppose there are N web pages
    2. Initialise: $r^{(0)} = [\frac{1}{N}, ..., \frac{1}{N}]^T$
    3. Iterate: $r^{t+1} = M \cdot r^{(t)}$
    4. Stop when $|r^{t+1} - r^t|_1 < \epsilon$
- **Random Walk Interpretation:**

Random Walk Interpretation

- Imagine a random web surfer:
  - At time $t = 0$, surfer starts on a random page
  - At any time t, surfer is on some page i
  - At time $t + 1$, the surfer follows an out-link from i uniformly at random
  - Process repeats indefinitely
- Let:
  - $p(t)$ ...vector whose ith coordinate is the prob. that the surfer is at page i at time t
  - So, $p(t)$ is a probability distribution over pages

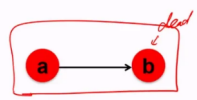
$p(t)$ where $t = 0$:



29

- Stationary Distribution: as $t \to \inf$, the probability distribution approaches a steady state, representing the long term probability that the random walker is at each node, which is the PageRank scores.
- Three Questions:
  1. Does the random walk converge? Not always
  2. Does it converge to what we want?
  3. Are results reasonable?
- Some pages are **Dead Ends**: Pages with no outgoing links. Causes importance to leak out of the network.

Problem 1: Dead Ends



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

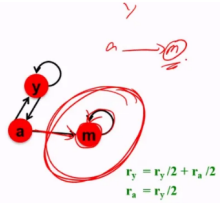
Example:

$$\begin{matrix} r_a \\ r_b \end{matrix} = \begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{matrix}$$

Iteration 0, 1, 2, ...

- **Spider Traps**: Pages that link to each other but have no outgoing links. Eventually, all importance will be trapped in the spider trap.

Problem 2: Spider Traps



- Power Iteration:
  - Set $r_j = 1/N$
  - $r_j = \sum_{i \to j} \frac{r_i}{d_i}$
  - And iterate
- Example:

$r_y = r_y/2 + r_a/2$
$r_a = r_y/2$
$r_m = r_a/2 + r_m$

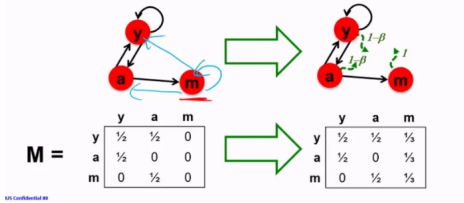
$$\begin{bmatrix} r_y \\ r_a \\ r_m \end{bmatrix} = \begin{matrix} 1/3 & 2/6 & 3/12 & 5/24 & \dots & 0 \\ 1/3 & 1/6 & 2/12 & 3/24 & \dots & 0 \\ 1/3 & 3/6 & 7/12 & 16/24 & & 1 \end{matrix}$$

Iteration 0, 1, 2, ...

- To solve deadends and spider traps, we cam introduce teleportation. At each step, the random walker has a small probability of teleporting to a random page. This ensures that the random walker can escape deadends and spider traps.
- PageRank Equation: $r_j = \sum_{i \to j} \beta \frac{r_i}{d_i} + (1 - \beta) \frac{1}{N}$
- β is also known as the damping factor. It is the probability that the random walker follows a link, and $1 - \beta$ is the probability that the random walker teleports.
- Google Matrix A: $A = \beta M + (1 - \beta)[\frac{1}{N}]_{N \times N}$
- PagreRank Equation (Matrix Form): $r = A \cdot r$
- In practice, β is usually set to 0.8 to 0.9. Which allows for 5 to 10 steps on average before teleport.
- If at a Dead End, Always Teleport: preprocess random walk matrix M and set each entry in the column of the dead-end page to $\frac{1}{N}$. This makes the matrix column stochastic.

Solution: If at a Dead End, Always Teleport

- Teleports: Follow random teleport links with probability 1.0 from dead-ends
  - Equivalently, for each dead end **m** we can preprocess the random walk matrix **M** by making **m** connected to ***every node*** (including itself).



Topic Specific PageRank

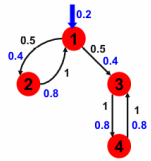
Problems with Simple Page Rank:
- Measures generic popularity of a page and does not consider popularity based on specific topics. Solution: Topic-Specific PageRank.
- Uses a single measure of importance. Solution: Hubs-and-Authorities.
- Susceptible to link spamming. Solution: TrustRank.
- Idea: Bias the random walk towards a specific topic.
- When random walker teleports, it picks a page from a set S.
- S contains only pages that are relevant to the topicFor each teleport set S, we get a different PageRank vector r_s.

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

- A is stoaachstic and column-normalised.

Example: Topic-Specific PageRank

$S = \{1\}$ $\beta = 0.8$



Node	Iteration 0	1	2	...	stable
1	0.25	0.4	0.28		0.294
2	0.25	0.1	0.16		0.118
3	0.25	0.3	0.32		0.327
4	0.25	0.2	0.24		0.261

Probabilities (no teleport)
Probabilities (with teleport)

S={1}, β=0.9: r=[0.17, 0.07, 0.40, 0.36]
S={1}, β=0.8: r=[0.29, 0.11, 0.32, 0.26]
S={1}, β=0.7: r=[0.39, 0.14, 0.27, 0.19]
S={1,2,3,4}, β=0.8: r=[0.13, 0.10, 0.39, 0.36]
S={1,2,3}, β=0.8: r=[0.17, 0.13, 0.38, 0.30]
S={1,2}, β=0.8: r=[0.26, 0.20, 0.29, 0.23]
S={1}, β=0.8: r=[0.29, 0.11, 0.32, 0.26]

56

- We can create different PageRank for different topics.
- Which topic ranking to use: User can pick from menu, classify query into topic, use context of query, user context.

PageRank Implementation

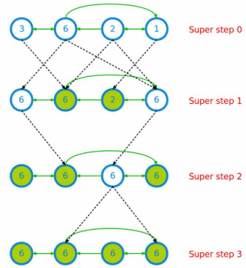
- For Graph Algorithms, it involves local computation at each vertex, and communication between vertices.
- **Think like a vertex:** Similar to MapReduce, the user only implements a function that is applied to each vertex.
- The framework abstracts away scheduling and implementation details.

Pregel Model

- Pregel is a distributed graph processing model developed by Google.
- It is based on the Bulk Synchronous Parallel (BSP) model.
- The computation is divided into **supersteps**. Each superstep consists of three phases: Computation, Messaging, Synchronization.
- **Computation:** Each vertex processes incoming messages and updates its state.
- **Messaging:** Vertices send messages to other vertices.
- **Synchronization:** All vertices synchronise and move to the next superstep.
- `Vertex.compute()` is the user-defined function that is called at each vertex in each superstep.
  - It can read messages sent to v in superstep $s - 1$.
  - It can send messages to other verices that will be read in superstep $s + 1$.
  - I can read or write the value of v and the values of its outgoing edges. Or even, add and remove edges.
- The computation is repeated until a termination condition is met.
  - A vertex can choose to deactivate itself
  - A vertex is "woken up" if it receives a message
  - Computation halts when all vertices are deactivated

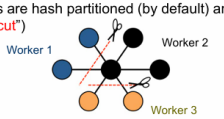
Example: Computing Max Value

```
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()
```



Pregel: Implementation

- Master & workers architecture
  - Vertices are hash partitioned (by default) and assigned to workers ("edge cut")



  - Each worker maintains the state of its portion of the graph in memory
  - Computations happen in memory
  - In each superstep, each worker loops through its vertices and executes `compute()`
  - Messages from vertices are sent, either to vertices on the same worker, or to vertices on different workers (buffered locally and sent as a batch to reduce network traffic)

66

- Fault Tolerance: Checkpointing to persistent storage, Failure detected through heartbeats, Corrupt workers are reassigned and reloaded from checkpoints.

PageRank in Pregel

```
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

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

Send outgoing messages

Stop after fixed no. of iterations

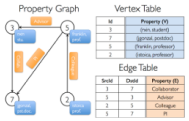
- **Note**: this algorithm implicitly assumes that the nodes' values (*MutableValue()) have been correctly initialized based on the initialization scheme that the user wants. In practice, you would generally do the initialization during superstep 0.

- Other Graph Processing Systems: Spark GraphX/GraphFrame (Extends RDDs to Resilient Distributed Property Graphs, uses vertex cut), Giraph, GraphLab, PowerGraph, Trinity, Pregel+, Neo4j.

Spark Implementation on weighted pagerank

- weighted page rank algorithm: PR = 0.15+0.85*sum(PR*weight)
- Stage 1: Join

```
CREATE VIEW triplets AS
SELECT s.Id, d.Id, s.P, e.P, d.P
FROM edges AS e
JOIN vertices AS s JOIN vertices AS d
ON e.srcId = s.Id AND e.dstId = d.Id
```



- Stage 2: Group-By

```
SELECT t.dstId, 0.15+0.85*sum(t.srcP*t.eP)
FROM triplets AS t GROUP BY t.dstId
```

- https://spark.apache.org/docs/latest/graphx-programming-guide.html