CS4225 Notes

by Zong Xun

Introduction

Big Data can be broken down into 4V's:

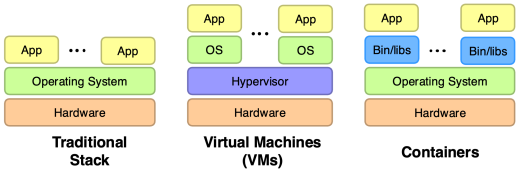
- **Volume**: Scale of Data
- **Velocity**: Streaming of Data
- **Variety**: Types of Data
- **Veracity**: Uncertainty of Data

Utility Computing

What & why?

- Computing resources as a metered service
- Ability to dynamically provision virtual machines
- Scalability: "Infinite" capacity
- Elasticity: Ability to scale down or up when required

How?



- **Virtual Machines:** enable sharing of hardware resources by running each application in an isolated virtual machine. *High overhead as each VM has its own OS.*
- **Containers:** enable lightweight sharing of resources, as applications run in an isolated way, but still share the same OS. A container is a lightweight software package that encapsulates an application and its environment.

Everything as a Service

- **IAAS:** Utility Computing. User rents a virtual machine and makes all decisions on what to run on it.
- **PAAS:** Provides hosting for web applications and takes care of hardware maintenance and upgrades.
- **SAAS:** Covers the entire stack, from infrastructure, to deployment to software.

Infrastructure: Data Centers

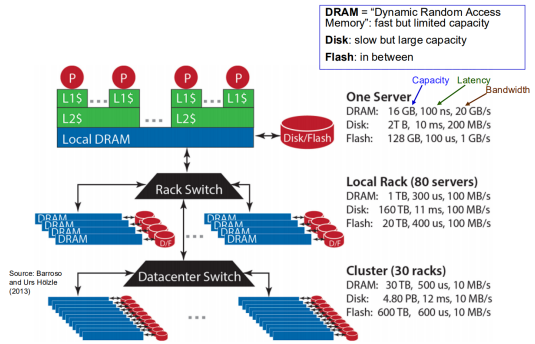
- **Single Server** – a commodity hardware
- **Rack** – contains dozens of servers, often connected to a rack switch (or top-of-rack switch)
- **Data center** – contains hundreds or more racks, connected to a data center switch (or core switch)

Bandwidth vs Latency

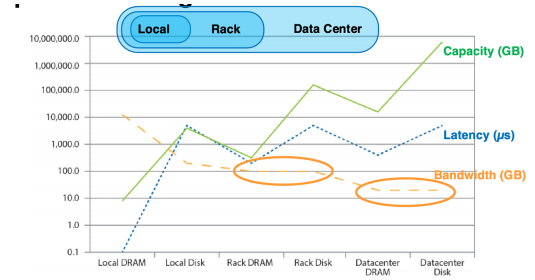
- **Bandwidth**: maximum amount of data that can be transmitted per unit time. Bandwidth of the whole path is approximately the *minimum bandwidth* along the path.
- **Latency**: time taken for *one* packet from src to dest (or rtt). Latency along a path *combines additively*. May not hold due to protocols, congestion and queueing etc.
- **Throughput**: Rate at which data was actually transmitted during some period of time.

Note: When transmitting (*large/small*) amount of data, (*bandwidth/latency*) tells us roughly how long the transmission will take.

Storage Hierarchy



Takeaways



- Capacity and latency increases, bandwidth decreases over storage hierarchy.
- Disk reads are much more expensive than DRAM, both in terms of higher latency and lower bandwidth.
- Latency is dominated by disk speed: since disk transfers have such high latency, any transfers involving disk take about the same amount of time.
- Bandwidth tends to be bottle-necked by networking switches: bandwidth decreases when we need to pass through these switches.
- Price and I/O speed decreases over the storage hierarchy.

Main ideas

- Horizontal scaling is preferred over vertical scaling.
- Clusters have limited bandwidth: we should **move processing to where data is stored**.
- Process data sequentially, avoid random access. Accessing data from disk involves "disk seeks" (e.g. spinning hard disk to a specific location) which leads to high latency. Disk throughput is reasonable when sequentially reading a lot of data.
- Increasing the number of processors should speed up the system without being disruptive. A (difficult) goal is "**linear scalability**": e.g. if we use 10x more machines, we can finish the job 10x faster.

MapReduce

Motivation: reading data using a single computer is too slow. Make use of distributed commodity nodes, and a distributed network (ethernet) to connect them. Adopt parallelization + Divide and Conquer paradigm.

Challenges

- Machine *Failures*.
- *Synchronization* is hard.
  - We don't know the order which workers run
  - When workers will interrupt each other
  - When workers need to communicate partial results
  - Need control mechanisms. **Barriers** ensure that every process must stop until all have reached the barrier.
- *Programming Difficulty*. Concurrency is difficult to reason especially at the scale of data-centers, in the presence of failures, and many interacting services.

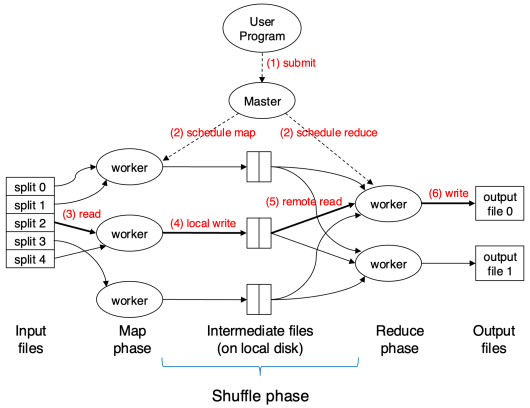
Data center is the computer

- Design for the **right level of abstraction**: user should not worry about unnecessary system level details (reliability, fault tolerance etc.)
- Separating the *what* from the *how*. Execution framework will take care of *how* to actually execute it.

Typical Big Data Problem

- Iterate over a large number of records
- Extract something of interest from each (**Map**)
- Shuffle and sort intermediate results (**Shuffle**)
- Aggregate intermediate results (**Reduce**)
- Generate final output

Map Reduce Implementation



1. **Submit**: Code and configuration to Master node
2. **Schedule**: Master schedules resources for map and reduce tasks (**does not handle any actual data**)
3. **Read**: Each split corresponds to one map task. **Workers execute one map task at a time**.
4. **Map phase**: Each worker iterates over each (key, value) tuple in its input split, and computes the map function.
5. **Local write**: Each worker writes the outputs of the map function to intermediate files on its **own local** disk. These files are partitioned into the chunks (**depends on configuration**) and sorted by key within each chunk.
6. **Remote read**: Each worker is responsible for one or more keys. For each key, the data is read from the **corresponding partition** of each mapper's local disk.
7. **Reduce phase**: The reducer receives all its needed key value pairs, where the keys arrive in **sorted** order. It computes the reduce function on the values of each key.
8. **Write**: The output is written to a distributed file system.

Note: "**Shuffle phase**" is comprised of local writes and remote read steps. It happens partly on the map workers, and partly on the reduce workers.

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

Data Transfer Overhead: Hadoop involves moving data across the network. This transfer incurs a baseline latency due to network bandwidth constraints, even for small data sizes. There's also overhead associated with the initialization and synchronization of tasks.

Detail: Barrier between map & reduce phases to ensure all data arrived in partition before remote read. Note that the shuffle phase can begin copying intermediate data earlier.

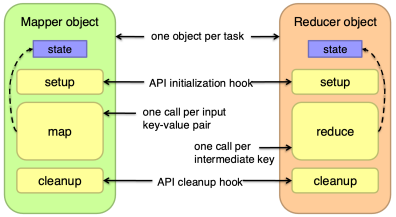
Extension

- The assignment of keys to reducers is determined by a hash function. A custom *partition* can better spread load among reducers.
- Reduce number of disk writes by locally aggregating the output from mappers. Whether the combiner runs **0, 1, or multiple times** should not affect the correctness of the final output. It must be both **associative** and **commutative** e.g. max, min, but not minus or mean.
- Note the **combiner** and **partitioner** runs after the map phase but before the local write.
- Optimal no.of tasks: better performance due to parallelism from 1 to n. From n to a certain point, better performance due to multi-tasking on the same machine. Afterwards, performance become stable or even degrade due to saturation.

Performance Guidelines for Basic Algorithmic Design

- Linear scalability: more nodes can do more work in the same time
- Minimize disk and network I/O; sequential vs. random; in bulk vs in chunks
- Reduce memory working set of each task/worker
- Large working set → high memory requirements / probability of out-of-memory errors.

Preserving state in Map / Reduce Tasks



Store state variables that can be shared across multiple map function calls within a map task. The `setup()` function is called only once per task before processing any keys.

Notes

1. #map function calls = #input key-value pairs

2. #map tasks = #input splits
3. #reduce function calls = #distinct intermediate keys
4. #reduce tasks = specified by the user

Secondary Sort

Define composite key as (K1, K2), where K1 is the original key and K2 is the variable we want to sort. Custom **Comparator**: compare by K1 first, then by K2. **Partitioner**: customize to partition by K1 only, not (K1, K2)

Note: May sort data in the reducer, but more expensive (in memory & time). Secondary sort is borrowing the inbuilt distributed sorting process of Hadoop to sort the tuples.

Hadoop Distributed File System

Assumptions

- Commodity hardware (best performance-cost ratio)
- High component failure rates
- Modest number of huge files
- Large **sequential** reads instead of random access
- Large batch reads and writes
- Not suitable for concurrent writes to same file

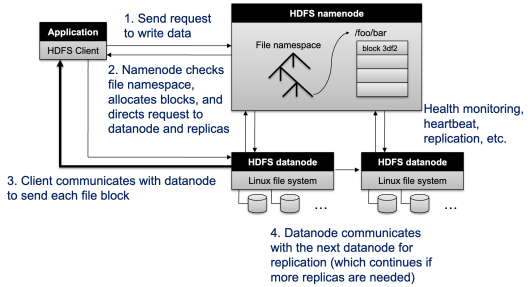
Design Decisions

- Files stored as chunks (or blocks)
- Reliability through replication - two secondaries. Slow write performance, and storage overhead. Robust.
- Single master to coordinate access, keep metadata
- Optimized for high throughput than low latency, well-suited for processing large datasets in batch mode.

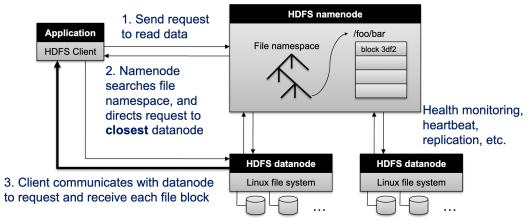
Interface

- Similar to interface of regular file systems, with extra commands to transfer files to and from local file-system.
- Designed for **write-once, read-many**. Does not support modifying files, other than appending.

Writing



Reading



Namenode responsibilities

Coordinating file operations
Managing file system name space

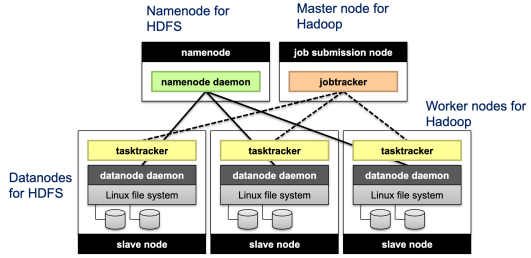
- Holds file/directory structure, metadata, file-to-block mapping, access permissions etc.
- Directs clients to datanodes for reads and writes.
- **No data is moved through the name node!**

Maintaining overall health

- "Heartbeat" (periodic communication with datanodes)
- Block rebalancing (ensure data is evenly distributed)

If the namenode's data is lost, there is no way to reconstruct the files on the filesystem from the raw block data. Fortunately, HDFS provides 2 ways of improving resilience, through backups and checkpointing.

Putting it together



Problem: High network traffic when moving data to compute nodes from storage nodes.

Solution: Same set of nodes are used. Hadoop schedule map tasks to run on the machines that already contain the needed data (**data locality**).

Relational DBs and MapReduce

Projection in MapReduce

- Map: take in a tuple (with tuple ID as key), and emit new tuples with appropriate attributes
- **No reducer needed** (No shuffle step, more efficient)

Selection

- Map: take in a tuple (with tuple ID as key), and emit only tuples that meet the predicate
- **No reducer needed**

Aggregation

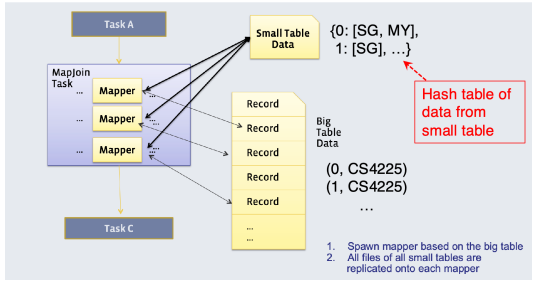
- Map over tuples, emit (key, value)
- Framework automatically groups these tuples by key
- Compute average in reducer
- Optimize with combiners

Relational Joins

Method 1: Broadcast (or 'Map') Join

Requires one of the tables to fit in memory.

- All mappers store a copy of the small table (**in memory**, with keys as the keys we want to join by).
- Iterate over big table, and join the records with small table.

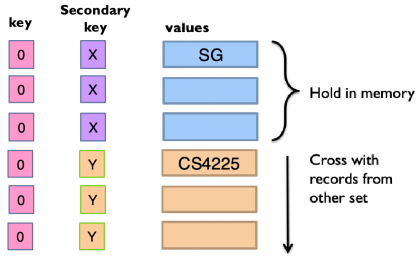


Method 2: Reduce-Side (or 'Common') Join

Generally, slower than broadcast join.

- Different mappers operate on each table, and emit records, with key as the variable to join by.
- In reducer: use secondary sort so 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.

In reduce function for key 0:



Similarity Measures

Lower distance = Higher similarity, and vice versa

- Euclidean Distance

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

- Manhatten Distance

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

- Cosine Similarity

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

Only considers direction i.e. *doesn't change when scaled*

- Jaccard Similarity

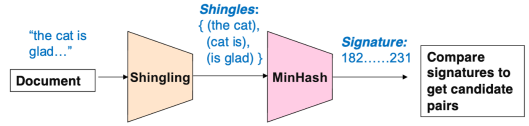
$$S_{Jaccard}(A,B) = \frac{|A\cap B|}{|A\cup B|}$$

- Jaccard Distance

$$d_{Jaccard}(A,B) = 1 - s_{Jaccard}(A,B)$$

2 Steps for Similar Docs

1. **Shingling:** Convert documents to **k-shingle** ≈ **k-gram**
2. **Min-Hashing:** Convert "shingles" to short "signatures". Documents with same signature are candidate pairs.



MinHash provides a **fast approximation** for pairwise Jaccard Similarities.

MinHash: Overview

Map each shingle to an integer and compute min.

1. $h(D)$ is small enough that the signature fits in RAM
2. highly similar documents have same signature with high prob. otherwise, $h(D_1) \neq h(D_2)$ with high prob.

Key Property: the probability that two documents have the same MinHash signature is equal to their Jaccard Similarity

$$Pr[h(D_1) = h(D_2)] = JaccardSim(D_1, D_2)$$

In practice, we usually use **multiple hash functions**, and generate N signatures for each document. Candidate pairs are defined as those matching a *sufficient number* (e.g. at least 50) among these signatures.

Implementation

- Map
  - Read over the document and extract its shingles.
  - Hash each shingle and take the min.
  - Emit (signature, document id).
- Reduce
  - Receive all documents with a given signature
  - Generate all candidate pairs from these documents
  - (Optional) check if the pairs are actually similar

Clustering

K-Means Algorithm

1. Initialization: Pick K random points as centers
2. Repeat:
   - **Assignment:** assign each point to nearest cluster
   - **Update:** new center = average of its assigned points
   - Stop if no assignments change

Naive Implementation (without combiners)

```
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!

This MapReduce job performs a single iteration of k-means.

Configure() reads in the current positions of the cluster centers (i.e. stars) from a file.

Map() assigns each point to the closest cluster.

Reduce() aggregates the points in each cluster.

Q: Disk I/O exchanged between the mappers and reducers? (let n = no. of points, m = no. of iterations, d = dimensionality, k = no. of centers) A: $O(nmd)$

Optimized Implementation

```
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.

In-mapper combiner is used. Disk I/O reduced to $O(kmd)$. High memory usage.

NoSQL

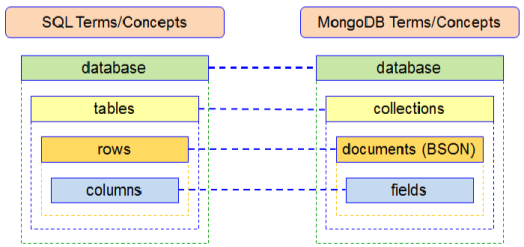
- Horizontally Scalability (Velocity/Vol.)
- Replicate/distribute data over many servers (Velocity/Vol.)
- Simple call interface
- Often weaker concurrency model than RDBMS
- Efficient use of distributed indexes and RAM
- Flexible schemas (Variety)

KV Stores

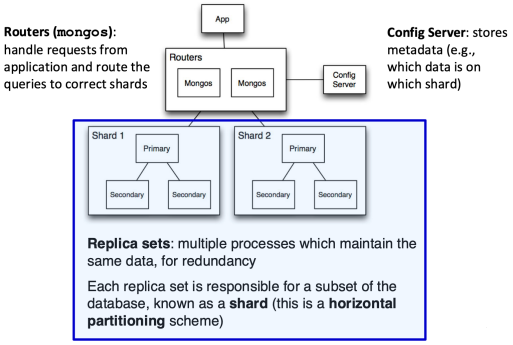
- Data Model
  - Stores associations between keys and values
  - Keys are usually primitives and can be queried
  - Values can be primitive or complex; cannot be queried.
- Operations
  - Simple API: GET and PUT, with optional operations like MULTI-GET, MULTI-PUT, and Range Queries.
  - Suitable for **fast reads and writes**, **storing basic information** and **if complex queries are not required**
- Implementation
  - Non-persistent; in-memory hash table
  - Persistent; stored persistently to disk

Document Stores

- A document is a JSON-like object: it has fields and values.
- Flexible schema: Documents **within the same collection** can have different fields. Allow adding new fields will not disrupt existing data.
- CRUD (create, update, read, destroy) – similar to SQL queries, but without fixed schema.
- Unlike (basic) key value stores, document stores allow querying based on the content of a document



Architecture of MongoDB



Routers (mongos): handle requests from application and route the queries to correct shards

Config Server: stores metadata (e.g., which data is on which shard)

Replica sets: multiple processes which maintain the same data, for redundancy

Each replica set is responsible for a subset of the database, known as a **shard** (this is a **horizontal partitioning** scheme)

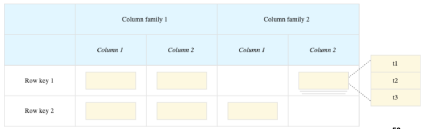
1. Query is issued to a router (mongos) instance
2. Config server to determine which shards to query.
3. Query is sent to relevant shards (partition pruning)
   - If the query is based on a key other than the shard key, it will go to all shards.
4. Shards run query on their data, and send results back.
5. Mongos merges the results and returns to the application.

Replication in MongoDB

- Standard configuration: 1 primary, 2 secondaries
- **Writes:**
  - The **primary** receives all write operations
  - Records writes onto its "operation log"
  - Secondaries replicate this operation log, apply to local copies (ensuring data is synchronized), then acknowledge the operation
- **Reads:**
  - The user can configure the "read preference", which decides if we can read from secondaries, or the primary
  - Allowing reading from secondaries can decrease latency and distribute load (improving throughput), but allows for reading stale data ("eventual consistency")

Wide Column Stores

- Rows describe entities
- Related columns are grouped as column families
- Sparsity: if column is not used, it doesn't use space
- Better compression, vectorized processsing but lower read efficiency when reading many fields.



Graph Databases

Use-cases crucially involve relationships e.g. recommending courses or content.

Vector Databases

- Store vectors (i.e row represents a point in d dimensions)
- Usually **dense**, **numerical** and **high-dimensional**
- Allow fast similarity search
- DB features: scalability, real-time updates, replication
- Commonly used to store embeddings (text/image)

Strong vs Eventual Consistency

- **Strong consistency**: any reads immediately after an update must give the same result on all observers
  - Readers are blocked until replication is complete.
- **Eventual consistency**: if the system is functioning, eventually all reads will return the last written value

ACID vs BASE

Relational DBMS provide stronger (ACID) guarantees, but many NoSQL system relax this to weaker "BASE" approach:

- Basically Available: basic reads and writes are available
- Soft State: without guarantees, we only have some probability of knowing the state at any time
- Eventually Consistent

Implications: eventual consistency offers better availability at the cost of a much weaker consistency guarantee.

- Systems are configurable for multiple different consistency levels (including strong) – 'tunable consistency'

Duplication

Some NoSQL databases do support joins (e.g. newer versions of MongoDB) but convention is to duplicate data to improve efficiency as storage is cheap. **Fast reads / writes.**

- Tables are designed around expected queries.
- Useful if queries are fixed, and few updates.
- Changes will need to be propagated to multiple tables.

Pros& Cons

Pros	Cons
+ **Flexible / dynamic schema**: suitable for less well-structured data	- **Less sophisticated query language**: SQL has highly optimized and advanced querying capabilities; NoSQL key-value stores lack complex querying abilities
+ **Horizontal scalability**	- **Weaker consistency guarantees / denormalization**: application may receive stale data that may need to be handled on the application side
+ **High performance and availability**: due to their relaxed consistency model and fast reads / writes	

Conclusion: whether denormalization is suitable; complexity of queries (joins vs simple read/writes); importance of consistency (e.g. financial transactions vs tweets); data volume / need for availability.

Distributed Databases

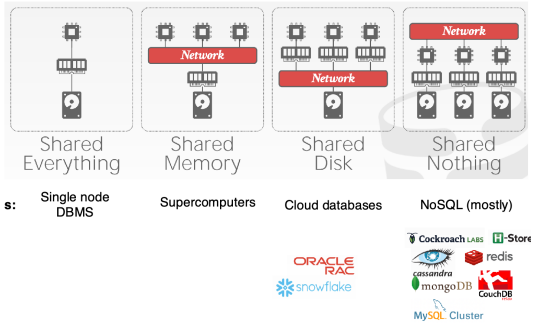
Why distributed?

- **Scalability**: allow database to scale by adding more
- **Availability / Fault Tolerance**: if one node fails, others can still serve requests nodes
- **Latency**: generally, each request is served by the closest replica, reducing latency, particularly when the database is distributed over a wide geographical area (e.g., globally)

Data Transparency

- Users should not be required to know how the data is physically distributed, partitioned or replicated.
- A query that works on a single node database should still work on a distributed database.

Distributed Database Architectures



Assumptions

- Nodes can go down (requiring the system to have sufficient redundancy)
- Network connections between some nodes may fail, causing the network to split into 2 parts which are completely cut off from each other. **Partition**.
- All nodes in a distributed database are well-behaved (i.e. they follow the protocol we designed for them; not adversarial or trying to corrupt the database)

CAP Theorem

A distributed system can have at most 2 of 3 properties.

- **Consistency**: Every read receives the most recent write.
- **Availability**: Every request to a node in the system will always lead to a response.
- **Partition Tolerance**: System continues to operate even when network failures cause a partition.

Data Partitioning

We can adopt **table partitioning**, by putting different tables on different machines. However, a table may grow too large, and still results in scalability concerns.

Horizontal Partitioning

Partition Key (or **shard key**) is the variable used to decide which node each tuple will be stored on. If we often need to filter tuples based on a column, then that column is a suitable partition key. Similar tuples are **co-located**.

Data from other partitions can be ignored (called **partition pruning**), which saves time as we don't have to scan these tuples. However, if there are too few unique keys (called **low cardinality**), this causes a lack of scalability. A similar problem occurs if some keys have too **high frequency**. Sometimes, these can be mitigated using **composite keys** (a combination of multiple keys). Tuples may need to be shifted between partitions often if the key changes.

Range Partition

Allow more efficient query and analysis on data within a specific time range. Within a specific time period, all the data are written to a specific partition, leading to the bottleneck issue at this specific partition server. Splitting the range is automatically handled by a balancer (it tries to keep the shards balanced)

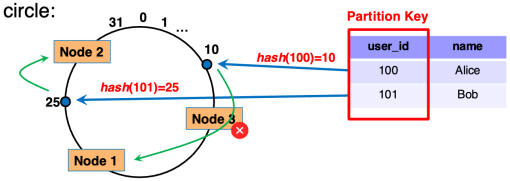
Hash Partition

Hash partition key, then divide that into partitions based on ranges. A well-designed hash function generates a uniform distribution that automatically spreads out partition key values roughly **evenly**. Not valid if there is skewed data, or disproportionate number of queries to a node.

Consistent Hashing

Weakness of previous approaches: in hash / range partitioning, if we want to partition over more nodes, or if a node goes down, a lot of data may have to be moved around, which is inefficient.

Think of the output of the hash function as lying on a circle:



- Each tuple is placed on the circle, and assigned to the node that comes clockwise-after it
- To delete a node, we simply re-assign all its tuples to the node clock-wise after this one
- To add a node, we add a new marker, and re-assigning all tuples that came before it to the new node

Simple replication strategy: replicate a tuple in the next few additional nodes clockwise after the primary node.

Multiple markers: we can have multiple markers per node. For each tuple, we still assign it to the marker nearest to it in the clockwise direction.
Benefit: when we remove a node, its tuples will not all be reassigned to the same node. This can balance load better.

Spark

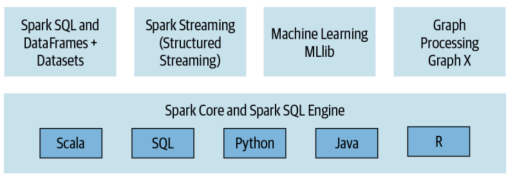
Spark stores **most** intermediate results in memory, making it faster, especially for iterative processing. When memory is insufficient, **spills to disk** which requires disk I/O.

- Speed, Ease of use, Modularity, Extensibility

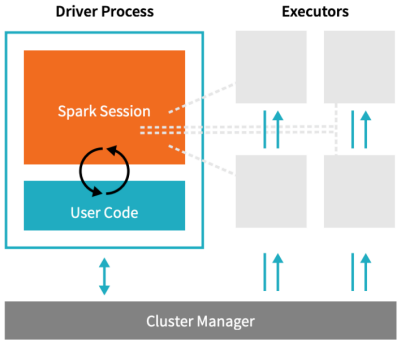
Hadoop

- **Network and disk I/O costs:** intermediate data is written to local disks and shuffled across machines (slow).
- Not suitable for **iterative** processing, as each individual step has to be modeled as a MapReduce job, with multiple writes to HDFS.

Spark Components and API Stack



Spark Architecture



- **Driver Process** responds to user input, manages the application and distributes work to **Executors**, which run code assigned to them and send results back to the driver.
- **Cluster Manager** (Kubernetes) allocates resources when the application requests it.
- In local mode, these processes run on the same machine.

Evolution of Spark APIs

- A collection of JVM objects
- Functional operators (map, filter, etc)

- A collection of Row objects
- Expression-based operations
- Logical plans and optimizer

- Internally rows, externally JVM objects.
- Almost the "Best of both worlds": type safe + fast

Resilient Distributed Datasets (RDDs)

- Achieve fault tolerance through lineages
- A collection of objects distributed over machines
- RDDs are **immutable**, main logical data unit.
- **Transformations** transform RDDs into RDDs. They are **lazy**, and will not be executed until an **action** is called on it. An **advantage** of lazy execution is that Spark can optimize the query plan to improve speed.
  - `map, order, groupby, filter, join, select`
- **Actions** trigger computation.
  - `show, count, save, collect`
- Transformations and actions are executed in parallel across worker nodes. Results are sent to the driver in the final step.

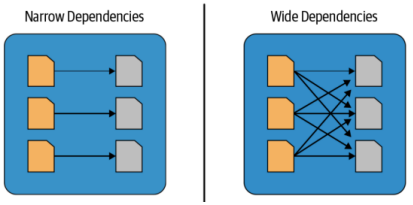
Caching

- `cache():` saves an RDD to memory (worker node).
- `persist(options):` can be used to save an RDD to memory, disk, or off-heap memory.
- **Cache when it is expensive to compute and needs to be re-used multiple times**.
- Note that if worker nodes does not have enough memory, they will evict the LRU RDDs. Be aware of memory limitations when caching.

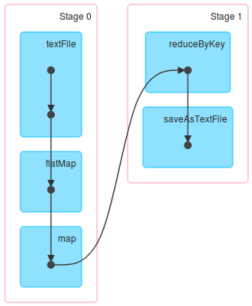
Directed Acyclic Graph (DAG)

- Internally, Spark creates a DAG which represents all the RDD objects and how they will be transformed.
- Transformations construct this graph; actions trigger computations on it.

Narrow and Wide Dependencies



- **Narrow dependencies** are where each parition of the parent RDD is used by at most 1 partition of the child RDD. E.g. `map, flatMap, filter, contains`.
- **Wide dependencies** are the opposite. E.g. `reduceByKey, groupBy, orderBy`.



- Consecutive **narrow** dependencies are grouped together.
- **Within stages**, Spark performs consecutive transformations on the same machines.
- **Across stages**, data needs to be shuffled, i.e. exchanged across partitions, in a process very similar to map -reduce, which involves writing intermediate results to disk

Lineage and Fault Tolerance

- Spark does not use replication to allow fault tolerance.
- **Spark tries to store all the data in memory, not disk.** Memory capacity is much more limited than disk, so simply duplicating all data is expensive.
- **Lineage approach**: if a worker node goes down, we replace it by a new worker node, and use the DAG to recompute the data in the lost partition. Only the RDDs from the lost parition is recomputed.

DataFrames (DF)

- Compared to RDDs, this is a higher level interface.
- DF operations are compiled down to RDD operations.

Comparison: RDD vs DF

RDD

- Instruct Spark **how** to compute the query
- Intention is completely **opaque** to Spark
- Spark does not understand the structure of the data in RDDs (which is arbitrary Python objects) or the semantics of user functions (which contain arbitrary code)

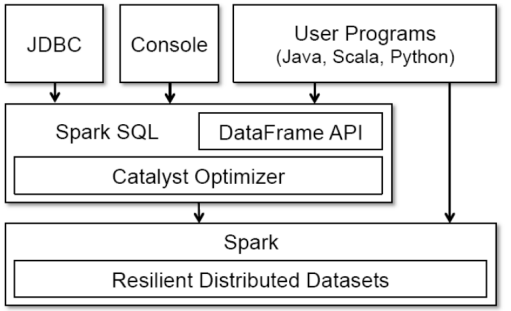
DataFrame

- Tell Spark **what** to do
- The code is far more **expressive** and **simpler**, using a domain specific language (DSL) and high-level DSL operators to compose the query
- Spark can parse this query and understand our intention, and then optimize the operations for efficient execution

Performance Bottleneck / Issues

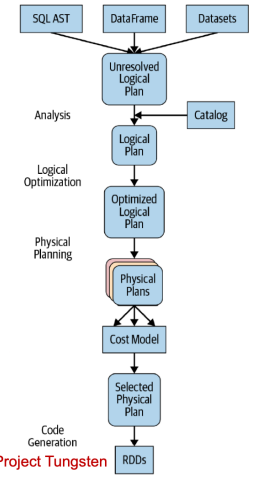
- Dataframes loaded in RAM, may incur potential I/O costs
- Wide dependencies may require data shuffling. It involves moving data between nodes, which incurs network and disk I/O overhead.
- Task straggling (some tasks may require more time, due to excessive load for a particular key) causing memory consumption issue and parallelism issue.
- Query too many results (add a limit)

Spark SQL



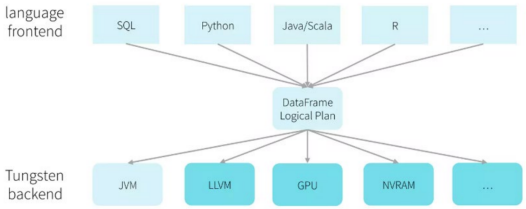
- **Unifies Spark components and permits abstraction to DataFrames in various programming languages**.

Catalyst Optimizer



Part of Spark SQL and responsible for transforming and optimizing SQL queries and DataFrame operations into an efficient execution plan through four transformational phases: (1) Analysis (2) Logical Optimization (3) Physical planning (4) Code generation (Tungsten)

Tungsten

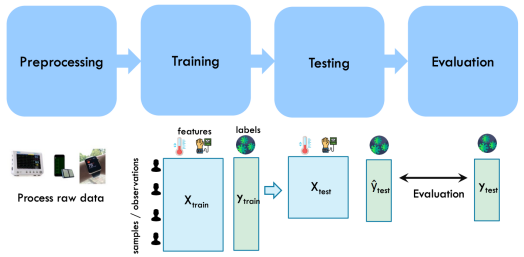


Unified API, One Engine, Automatically Optimized

- Substantially improve memory and CPU efficiency of Spark applications
- Push performance closer to the limits of modern hardware
- Memory Management and Binary Processing, Cache-aware computation, and code generation

Machine Learning with Spark ML

Typical Machine Learning Pipeline



Big Picture: machine learning involves fitting the parameters of a model (weights and bias) by minimizing the loss function.

Missing data

Why is data missing?

- Information was not collected
- Missing at random: missing values are randomly distributed. If data is instead missing not at random: then the "missingness" itself may be important information.

How to handle?

- Eliminate objects (rows) with missing values
- Fill in the missing values ("imputation") based on the mean / median of that attribute or by fitting a regression model to predict that attribute given other attributes
- **Dummy variables**: optionally insert a column which is 1 if the variable was missing, and 0 otherwise

Categorical / One Hot Encoding

- Convert categorical feature to numerical features. Numerical values should reflect **ordinal** or **inherent** order among categories.
- Convert discrete features to a series of binary features. Useful if there's no **ordinal** relationship.

Normalization

Normalize numeric features, so that scale does not affect regression results. For example, *clipping*, *log transform*, *standard scaler* and *min max normalization*.

Evaluation

Common measures

- Accuracy = (TP + TN) / (TP + FP + TN + FN)
- Precision = TP / (TP + FP)
- Recall = TP / (TP + FN)
- F1 score = 2 x (Precision x Recall) / (Precision + Recall)

Errors

- MSE = $\frac{\sum(y_i - y'_j)^2}{n}$
- Root MSE = $\sqrt{MSE}$
- MAE = $\frac{|y_i - y'_j|}{n}$
- R^2 higher the better. The percentage of variance from the samples can be explained through the model.

Pipelines

A pipeline chains together multiple Transformers and Estimators to form an ML workflow.

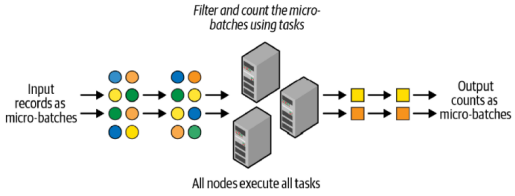
- **Transformers** are for mapping DF to DF. They have a `transform()` method. Append result to original DF.
- **Estimator** takes in data, and outputs a fitted model. They have a `fit()` method, which returns a Transformer.
- **Model training stage** iterative distributed in-memory computation. Cache training data in memory across iterations. Use broadcast variable to save & broadcast weights iteration by iteration

Stream Processing

Process data as received (from **input ports**), unlike offline or batch approaches that operate on full dataset.

- Stream is potentially infinite; the system **cannot store the entire stream accessibly** (due to limited memory)
- **High volume** and **constantly arriving over time**

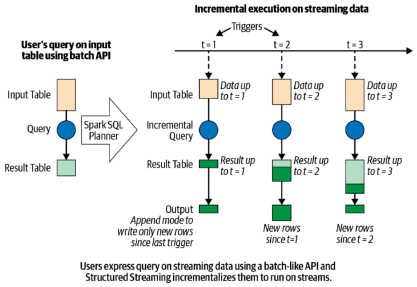
Spark Micro-Batch Stream Processing



Micro-batch processing breaks a continuous stream into small, finite batches of data that are processed **sequentially**. Each batch is processed in a **distributed** manner and computed **atomically** before moving on.

- Advantages
  - Fault tolerant; Efficiently recover from failures
  - **Exactly once** processing guarantees (checkpointing and write-ahead logs, idempotent sinks and transactional writes)
- Disadvantages: Latencies of a few seconds
  - Depend on the sink's support for idempotent writes
  - **Inherent** delay (intervals, coordination, and shuffling)

Philosophy of Structured Streaming



- Single, unified programming model and interface for batch and stream processing
- Data stream as an **unbounded** table
- **Incremental Execution**

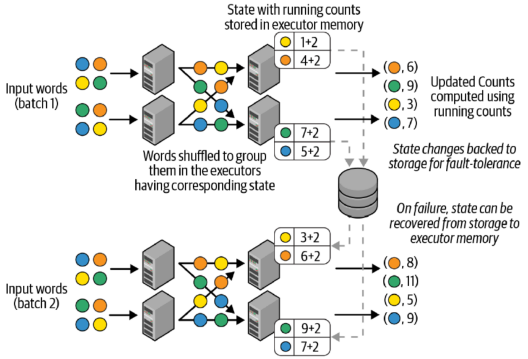
Five Steps to Define a Streaming Query

1. Define input sources
2. Transform data
3. Define output sink and output mode
   - Output writing details
   - where and how to write the output
4. Specify processing details
   - **Triggering details**: when to trigger the results and processing of newly available streaming data.
   - **Checkpoint Location**: store the streaming query process info for failure recovery
5. Start the query

Stateful Streams

- Not just perform trivial record-at-a-time transformations `(map, filter, select)`
- The ability to store and access intermediate data (program variables, local files or embedded or external databases).
- The state is maintained in the memory of the Spark executors and is check-pointed to the configured location to tolerate failures (both local and persistent storage)

Failure Recovery



Checkpoints are stored in a reliable distributed file system (e.g., HDFS or S3) for fault tolerance.

Recovering from Executor Failures

- Spark uses the checkpointed state to restore the lost partition of the state on another executor.
- State updates from the Write-Ahead Log (WAL) are replayed to ensure no updates are lost.
- The tasks are rescheduled on available executors, and processing resumes from the latest checkpoint.

Recovering from Driver Failures

- The driver stores the stream's execution progress and controls the execution plan.
- New driver reads the checkpointed progress and state.
- Processing resumes from last successfully completed batch.

Recovering In-Progress Micro-Batches

- Spark rolls back partially applied changes using the WAL.
- The batch is reprocessed from the checkpointed offsets and state.

Aggregations Not Based on Time

- Global aggregations `DF.groupBy().count()`
- Grouped aggregations `DF.groupBy('id').mean('value')`
- All built-in aggregation functions in DF are supported `sum(), mean(), stddev(), countDistinct(), collect_set(), approx_count_distinct()`

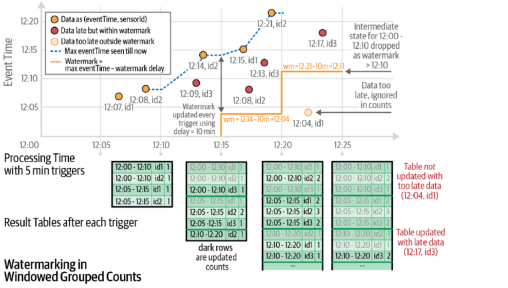
Time Semantics

Prefer *event-time* over *processing-time*

- Event time completely decouples the processing speed from the results (in case of system delays).
- Operations based on event time are **predictable** and results are **deterministic**.
- An event time window computation will yield the same result no matter how fast the stream is processed or when the events arrive at the operator.
- Watermarks: how long to wait before we can be certain that we have received all events that happened before a certain point in time.

Aggregations Based on Time

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

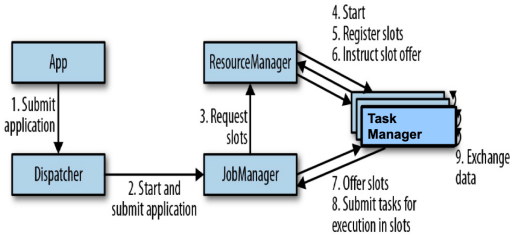


Performance Tuning

Besides tuning Spark SQL engine, other considerations

- Cluster resource provisioning appropriately to run 24/7
- No. of partitions for shuffles is **lower** than batch queries
- Setting source rate limits for stability
- Multiple streaming queries in the same Spark application

Flink

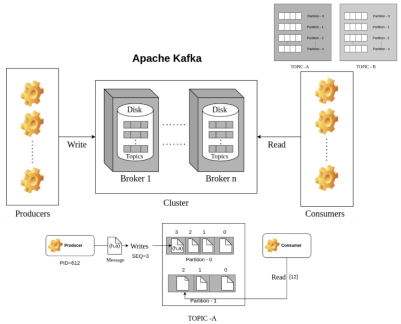


A distributed system for stateful parallel data stream processing. Flink is designed for event-time and processing-time stream processing, ensuring low-latency reactions to incoming events.

Dataflow Model

In a dataflow graph, the nodes represent the operations and edges denote data dependencies. Data enters the system through input sources, such as **event logs**, databases, or external streams.

Kafka

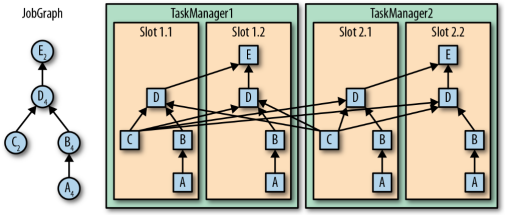


Kafka is a high-throughput, fault-tolerant, and scalable data pipeline system that uses a **publish-subscribe model**. Producers send messages to **topics**, which are partitioned and replicated across a cluster for scalability and fault tolerance. Kafka's **log-based storage** ensures message durability and replayability, while its design **decouples producers and consumers**, **enabling asynchronous and non-blocking communication**. Consumer groups allow parallel processing with guaranteed message ordering within partitions.

Data Warehouse vs. Streaming

Data Warehouse	Streaming Application
Batch-oriented	Real-time, continuous
High Latency (due to ETL)	Low Latency
Multi-step process incl. ETL, data warehousing, and reporting tools	Unified, streamlined pipeline combining ingestion, processing, and state management
Complex (Multiple DBs + ETL + Warehouse)	Simpler (single stream processor)

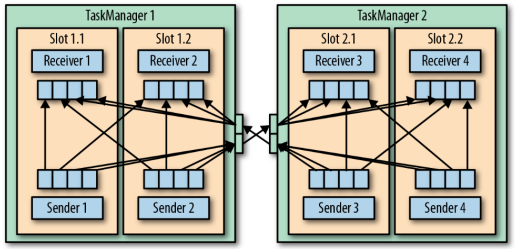
Task Execution



A **TaskManager** offers a *#processing slots* to control *#tasks* it is able to concurrently execute. A **processing slot** can execute one slice of an application - one parallel task of each operator of the application.

- tasks of the same operator (data parallelism)
- tasks of different operators (task parallelism)
- tasks from a different application (job parallelism)

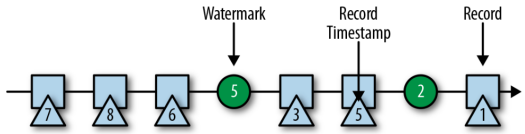
Data Transfer



- Tasks are continuously exchanging data.
- The TaskManagers take care of shipping data from sending tasks to receiving tasks.
- The network component of a TaskManager collects records in **buffers** before they are shipped.

Event Time Processing

- 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. Watermarks flow in a stream of regular records with annotated timestamps.
- Possible for out-of-order / latecomers to happen. Will be handled by the late-handling function.



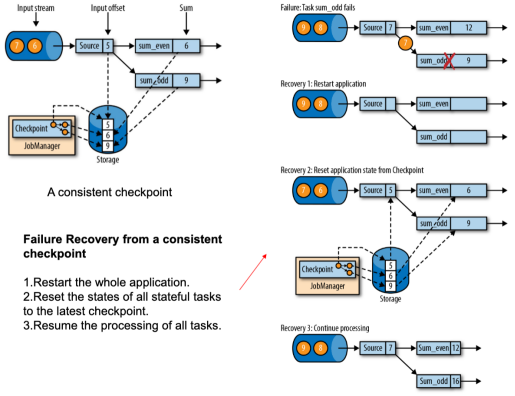
State management

- Local State Management
  - a task of a stateful operator reads and updates its state **locally** for each incoming record to ensure fast state accesses.
- Checkpointing
  - A TaskManager process may fail at any point, hence its storage must be considered volatile
  - The remote storage for checkpointing could be a distributed filesystem or a database system

Checkpoints

Consistent checkpoints (similar to spark)

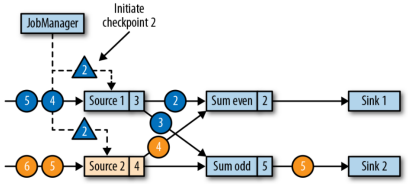
1. **Pause** the ingestion of all input streams
2. Wait for all in-flight data to be completely processed, meaning all tasks have processed all their input data
3. Take a checkpoint by copying the state of each task to a remote, persistent storage. The checkpoint is complete when all tasks have finished their copies.
4. Resume the ingestion of all streams.



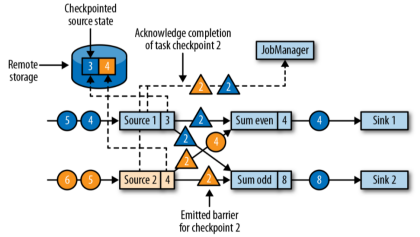
Flink's Checkpointing Algorithm

- Does not pause the complete application but decouples checkpointing from processing
- Tasks continue processing while others 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
- Flink's distributed checkpointing mechanism doesn't rely on buffering all the data in memory. Instead, it periodically captures the state of the computation and persists it to durable storage, ensuring exactly-once semantics through careful management of state.

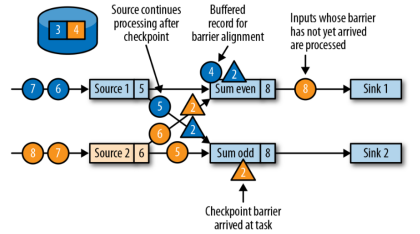
Streaming application with two stateful sources, two stateful tasks, and two stateless sinks. JobManager initiates a checkpoint by sending a message to all sources.



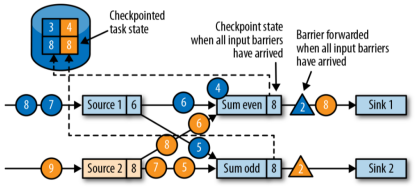
Sources checkpoint their state and emit a checkpoint barrier



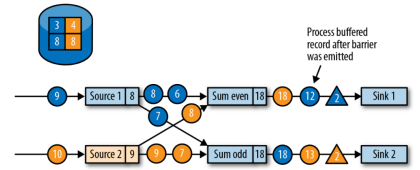
Tasks wait to receive a barrier on each input partition. Records from input streams for which barrier already arrived are buffered. All other records are regularly processed.



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.



A checkpoint is complete when all tasks / sinks have acknowledged the successful checkpointing of their state.

Comparison (Spark vs Flink)

Spark

- Microbatch streaming (latency of a few seconds)
- Checkpoints are done for each microbatch in a synchronous manner ("stop the world")
- Watermark: to determine when to drop the late events

Flink

- Real-time streaming (latency of milliseconds)
- Checkpoints are done distributed in an asynchronous manner (more efficient ⇒ lower latency)
- Watermark: a special record to determine when to trigger the event-time related results
- Flink uses **late handling functions** (related to watermark) to determine when to drop the late events

Graphs and PageRank

Graphs are made up of **entities** and **relationships**.

Simplified PageRank

- **Idea**: Links as votes. Page is more important if it has **more in-links**. Naive, as malicious user can create a huge number of "dummy" web pages, to link to their page, to drive up rank.
- **Solution**: make the number of 'votes' that a page has proportional to its own importance. If "dummy" pages themselves have low importance, they will contribute little votes. **Links** from **important** pages counts **more**.

Matrix Formulation

If page j with importance r_j has d_j out-links, each link gets r_j/d_j votes.
Define importance or "rank" vector where

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

where d_i is the number of out-links from page $i \to j$ and the sum of the vector, $\sum_i r_i = 1$.
Define the **Stochastic Adjacency Matrix**, M, where
- If $i \to j$, then $M_{ji} = \frac{1}{d_i}$ else $M_{ji} = 0$.
- Row vector has no interpretation
- Column vector **sum up to one**.

The flow equation can be written as

$$r = M \cdot r$$

Using **Power Iteration**,
- Suppose there are N web pages
- Initialize: $r^{(0)} = [1/N, \ldots, 1/N]^T$
- Iterate: $r^{t+1} = M \cdot r^t$
- Stop when L1 Norm $|r^{(t+1)} - r^{(t)}|_1 < \epsilon$

Interpretation: each node starts with equal importance (of 1/N). During each step, each node passes its current importance along its outgoing edges, to its neighbors.

Random Walk Formulation

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
- Repeat indefinitely

Let $p(t)$ be a vector whose ith coordinate is the probability that the surfer is at page i at time t. So, $p(t)$ is a probability distribution over pages.

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

Notice the two formulation are the **same**.
Stationary Distribution: as $t \to \infty$, the p.distribution approaches a **steady state**, representing the long term probability that the random walker is at each node, which are the PageRank scores.

PageRank with Teleports

Problems: (1) with the wrong **initial** values, PageRank may not converge. (set all to $1/N$) (2) Even if converge, may not be representative.

Deadends

Some pages have dead ends, which cause importance to "leak out". Have no out-links.

Spider traps

Random walk gets "stuck" in a trap, and eventually spider traps absorb all importance. All out-links are within group.

Teleports

At each time step, the random surfer has two options:
1. With probability β, follow a link at random
2. With probability $1 - \beta$, jump to some random page where $0.8 \leq \beta \leq 0.9$.

PageRank equation

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

where

$$A = \beta M + (1 - \beta)[\frac{1}{N}]_{N \times N}$$

hence, we have $r = A \cdot r$
Note: For each dead end, **m**, we **preprocess** the matrix by making **m** connected to every node **in the topic set**, including itself with prob. $1/N$.

Problems

- Measures generic popularity of a page
- Use a single measure of importance
- Susceptible to Link Spam

Topic-Specific PageRank

Measure popularity within a topic. Random walk has a small probability of teleporting at any step. Teleport can go to:
- **Standard PageRank**: Any page with equal probability.
- **Topic Specific PageRank**: A topic-specific set.

Idea: Bias the random walk. When random walker teleports, it picks a page from the set S. S contains only pages that relevant to the topic. **For each teleport set S, we get a different vector** r_s.

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

PageRank Implementation

Characteristics of Graph Algorithms

Common features of graph algorithms:
- Local computations at each vertex
- Passing messages to other vertex

Think like a vertex: algorithms are implemented from the view of a single vertex, performing one iteration based on messages from its neighbor.
- User only need to implement `compute()`.
- The framework abstracts away the scheduling / implementation details.

Pregel: Computational Model

- In each **superstep**, the framework invokes a user-defined function, `compute()`, for each vertex (conceptually in parallel)
- `compute()` specifies behavior at a single vertex v and a superstep s:
  - It can **read** messages sent to v in superstep $s - 1$
  - It can **send** messages to other vertices that will be read in superstep $s + 1$
  - It can **read or write** the value of v and the value of its outgoing edges (add or remove edges)
- This allows for efficient distributed computation by enabling each vertex to independently execute computation logic and communicate with its neighbors via messages. This greatly reduces the need for global coordination and enhances scalability and performance in large-scale graph processing.
- **Termination**:
  - A vertex can choose to deactivate itself
  - Is "woken up" if new messages received
  - Computation halts when all vertices are inactive

```
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**)

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

```
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.

Extra

Geographically distributed Data centres
- the network latency tends to increase and the network bandwidth tends to decrease, as the distance between two data centers increases.
- Extend the combiner to geo data center.
- HDFS Replication adaptation for geo-reliability (two within the data center, and one on another data center).

Replication Strategies
- Two replicas in the same rack, and one in the other rack.

Relational Tables and Spark
- Schema Support
- Query Optimization
- High level languages

Column Store
- Vectorized Processing
- Better Compression
- Operate directly on Compressed Data
- **Does not have better read efficiency when reading many fields**

High-Speed Data Center Network vs Advanced CPUs with much more cores and higher clock frequency
- Less aggressive **In-Mapper Combiner**
- Bring data to compute
- More parallelism
- Reduce chunk size (better compression)

Miscellaneous
- Performance **bottleneck** is different from performance **slow-down**. The former refers to baseline latency as a result of inherent design decisions.
- Without proper data distribution strategies, data partitioning in NoSQL databases can result in uneven data distribution, leading to hotspots and imbalanced resource utilization among nodes.
- All the languages will achieve similar performance through DataFrame API. API / Query language only provides a logical plan, and Catalyst Optimizer will generate the optimized logical plan, physical plan and RDD codes, which is the same no matter which API language you are using.
- Teleport is designed to solve the spider trap and dead ends problems in PageRank algorithm. Teleport helps the random walker to occasionally jump to the less popular /important nodes so as to spread out the importance factors among all the nodes in the graph. This helps the algorithm to converge faster.