Check

- ☑ ~~MapReduce word count specific example~~
- ☑ ~~RDDs creation (scala) and operations chapter 6~~
- ☑ ~~Structured data scala chapter 7~~
- ☑ ~~BigTable how it works~~
- ☑ ~~Kafka how it works~~
- ☑ ~~CAP and which tools uphold which~~
- ☑ ~~Resource allocation calculations~~
- ☑ ~~How does Cassandra work, Neojs~~
- ☑ ~~Pregel, GraphLab, and PowerGraph Graphs~~

Important

- Describe all operations on GFS such as read, write, append, delete
- Different types of MapReduce actions such as word count, matrix multiplication, etc.
- Describe different types of join such as reduce-join, map-side join

Last Summary

- Chubby
    - Discover new tablets servers
    - Make sure that their exists only one active master
    - Elects new master if previous one fails
- SSTable
    - Efficient lookups
    - Range queries
- Kafka
    - Producer: Sends messages to Kafka.
    - Consumer: Reads messages from Kafka.
    - Topic: A category or feed name to which records are published by the producers.
    - Partition: Topics are split into partitions, which are the basic unit of parallelism in Kafka.
    - Broker: A Kafka server that stores data and serves client requests.
    - Cluster: A group of brokers working together.
    - Ordered and append only
    - Log centric
- Other broker systems
    - Message queue
    - Message centric
- CAP
    - BigTable: C_P
    - Cassandra: _AP
    - Neo4j: CA_

- emit(key, value)
- What are the similarities and differences among Mesos (two level scheduling), YARN, and Borg (monolithic schecular)?
- Data warehouse (SQL query), Data lake, Data lakehouse
    - Conccurent writes using transaction log, delta protocol and ACID
- DStream
    - Processing real-time streams, DStreams represent a continuous stream of data, either generated by reading data from sources like Kafka, HDFS
    - Stataless operators scuh as map gets applied independently to each element that arrives
    - Statefull operators such as UpdateStaetByKey allows maintain state while continuously updating it with new information. It will allow access for previous states and new values for a specific key
- MapWithState vs updateStateByKey
    - updateStateByKey aggregate state across Dstreams, requires entire state to be stored in memory
    - mapWithState allows maintain state and return transformed data streams (is better)
- Datatframe, DataSet, RDD
    - Dataframe: Spark's Catalyst, allows SQL queries, less Type safe
    - DataSet: Catalyst, allows sql queries, better type safe, catch errors at compile time
- MapReduce skewed
    - Skew detection
    - key destributiomn
    - custom partitioning schemas
- GFS undefined
    - Partial writes
    - concurrent writes
    - chunk server failures
    - master server failures
- GFS Primary node
    - Hadnle READ requests
    - Data consistency
    - replica coordinate
- Spark failure
    - Task retry default 4 times
    - node failure running tasks moved to another node
    - data persistence and caching to retrieve data
    - Staggering handling
- Narrow vs Wide
    - Narrow transformations: These transformations do not require data shuffling between partitions. Examples are maps, filters, union etc.
    - Wide transformations: These transformations require data shuffling between partitions. Examples are groupByKey, reduceByKey, join, etc
- Graphs
    - Pregel: Vertex centric, superstep, designed for scalability and fault tolerance

        - GraphLab: Vertex centric, asynchronous execution, designed for performance and dynamic graph algorithms
        - X-stream: Edge centric, handle very large graphs, designed for scability
        - Powergraph: Edge centric
- Messages passing:
    - A Distributed model where each node exists on different machines
    - Each node has its own memory and processing power
    - Communicate by sending and receiving messages
- Shared Memory:
    - All nodes in a unified memory space
    - Threads can access any part of the graph without sending and receiving messages

Definitions

Distributed File Systems:

Understand the principles and components of distributed file systems.
Familiarize yourself with concepts like data replication and fault tolerance.

NoSQL Databases:

Learn about different types of NoSQL databases (e.g., document-oriented, key-value, column-family, graph).
Understand data modeling in NoSQL databases.

Scalable Messaging Systems:

Study the characteristics of messaging systems used for data processing and communication between components.

Big Data Execution Engines:

Get a strong grasp of MapReduce and Apache Spark.
Understand how these frameworks process large datasets.

High-Level Queries and Interactive Processing:

Study tools like Hive and Spark SQL for querying and analyzing data in a distributed environment.

Stream Processing:

Learn about stream processing frameworks and their applications.

Graph Processing:

Understand graph processing platforms and algorithms for analyzing graph data.

Scalable Machine Learning:

Explore machine learning in a distributed context, including algorithms and tools for large-scale data.

Resource Management:

Study resource allocation and scheduling in distributed computing clusters.

Large-Scale Distributed Systems:

Understand the architecture and properties of systems for storing and processing massive data.

Computational Models:

Be familiar with batch processing and stream processing models.
Analytics on Massive Data:

Learn how to design and implement nontrivial analytics on large datasets.
Scheduling and Resource Allocation:

Understand the models for scheduling and allocating computational tasks on large clusters.
Efficient Algorithms for Distributed Computing:

Explore the trade-offs involved in designing efficient algorithms for distributed data processing.
Based on the old exam questions, here's a more specific breakdown of topics to focus on:

MapReduce:

Understand the MapReduce framework and its phases (map and reduce).
Know how data is processed and how output is generated.

HDFS/GFS:

Understand how distributed file systems (HDFS or GFS) are used in MapReduce jobs.
Consistent Hashing:

Know the concept of consistent hashing and how it can distribute data across a network.

Spark:

Learn about Spark's lineage graph and the difference between narrow and wide transformations.
Understand how Spark handles joining tables of different sizes.

Streaming:

Understand how Storm can be used for stream processing and the components like spouts and bolts.
Compare fault tolerance models in Storm, Spark Streaming, and Flink.

Graph Processing:

Differentiate between message passing and shared memory models in graph processing platforms.
Write pseudo-code for a vertex-centric Gather-Apply-Scatter operation.

Summary

https://id2221kth.github.io/

1. Introduction

https://id2221kth.github.io/slides/2023/01_introduction.pdf

NIST (National Institute of Standards and Technology)

NIST is defined by:

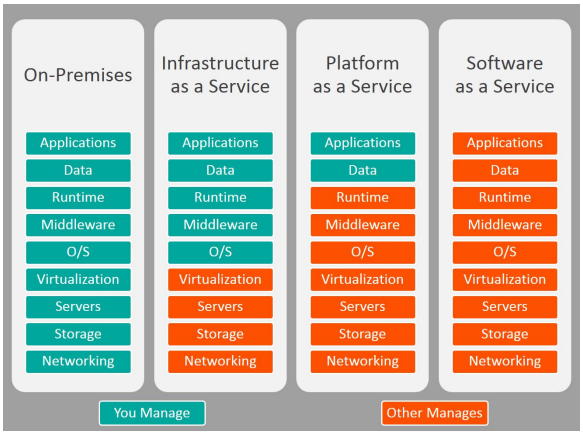
- Five characteristics
- Three service models
- Four deployment models

Cloud Characteristics

- **On-demand self-service:** A consumer can independently provision computing capabilities
- **Ubiquitous Network Access**: available for any network devices over the network
- **Resource Pooling:** Providers computing resources are pooled to serve customers
- **Rapid Elasticity:** Can rapidly scale on demand.
- **Measured Service:** Can easily measure control and report statistics of resource usage

Cloud Service Models

- **(Software as a Service) Saas:** It is like living in a hotel where vendors provide application access over a network like Gmail and GitHub.
- **(Infrastructure as a Service) IaaS:** Like building a new house. A vendor provides resources that a consumer can utilize, often via a customized virtual machine like EC2 or s3.



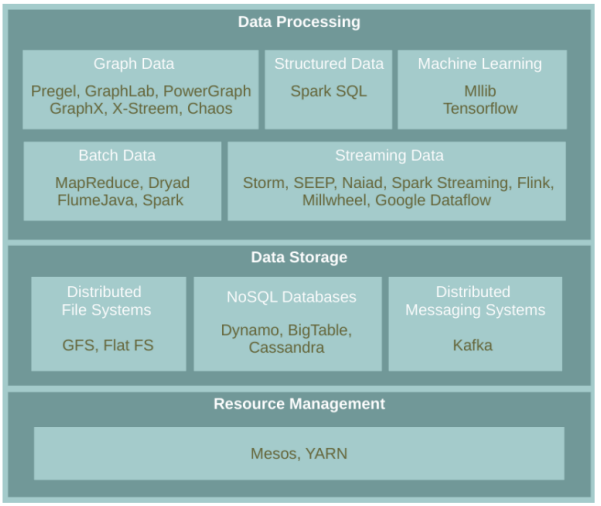
- **(Platform as a Service) PaaS:** Building an empty house. The vendor provides hardware and development environments like the Google app engine.

Deployment Models

- **Public Cloud:** AWS; Azure, GCP, etc. The main services that they provide are computing power, storage, databases, and big data analytics.
    - **Storage:** File storage, block storage, and object storage
    - **Big Data Analytics:** Data warehouse, streaming queueing.

Big Data

- Big data is characterized by 4 key attributes: **volume, variety**, **velocity**, and **value**.
- Scaling
    - Traditional platforms fail due to performance and need a new system that can store and process large-scale data
    - **Scale vertically** (up) by adding resources to a single node in a system. Usually more expensive than scaling out.
    - **Scale horizontally** (out) is by adding more nodes to a system. Usually more challenging due to fault tolerance and software development



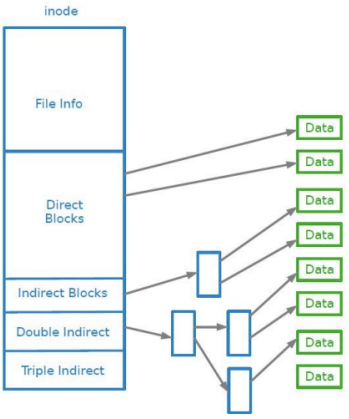
- **Resource Management:** Manage resources of a cluster
- **Distributed file systems:** Store and retrieve files in/from distributed disks
- **NoSQL databases:** BASE instead of ACID
- **Data Storage:** Store streaming data
- **Batch Data:** Process data-at-rest, data-parallel processing model
- **Streaming data:** process data-in-motion
- **Linked data (Graph data):** graph parallel processing model, vertex-centric and edge-centric programming model
- **Structured data**
- **Machine Learning:** Data analysis supervised and unsupervised learning

2 & 3: Storage

https://id2221kth.github.io/slides/2023/02_storage_part1.pdf
https://id2221kth.github.io/slides/2023/03_storage_part2.pdf

File System

- Controls how data is **stored** in and **retrieved** from a storage device
- Distributed is used when data **outgrows** the storage capacity of a single machine and is required to use a partition of several separate machines



Google File System (GFS)

Motivation and Assumptions

- Huge files (multi-GB)
- Files are usually modified by appending to the end meaning that **random writes** are non-existent
- Optimize for streaming access
- Node **failure** happens frequently

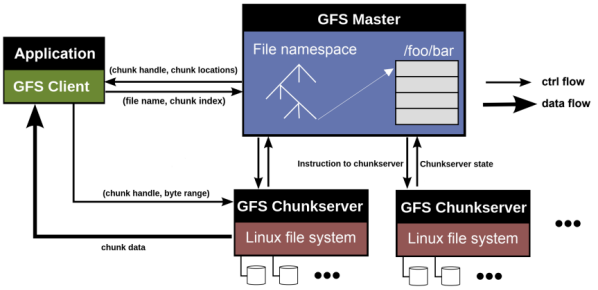
Optimized for streaming

- This means write once and read many (uses an append write system and successive read (like streaming).

Files and Chunks

- Files are split into different chunks
- A chunk is an immutable and globally unique chunk handle. Each chunk is stored as a plain Linux file

GFS Architecture



- **GFS Master**
  - Responsible for all system-wide activities
  - Maintains all file system metadata such as namespaces, mapping from files to chunks, locations, etc. Everything is kept in memory and is persistently in operation log
  - It periodically communicates with each chunk server to determine the location and overall state of the system (health)
- **GFS Chunkserver**

    - Manage chunks
    - Tells master what the chunks it has
    - stores chunks as files
    - maintains **data consistency** of chunks
- **GFS Client**
    - Issues **control requests** to master
    - Issues **data requests** directly to chunk servers
    - Caches metadata but does not cache data
- **Data Flow and Control Flow**
    - Data flow is decoupled from the control flow
    - This means the client **interacts with the master for metadata operations (CF)**
    - The client **interacts with chunk servers for file operations / data (DF)**
- **Why Large Chunks?**
    - 64MB or 128MB
    - **Advantage:** Reduces the size of the metadata stored in the master
    - **Advantage:** Reduces client's need to interact with master
    - **Disadvantage:** Wasted space due to internal fragmentation

System Interface

- Supports typical file operations such as create, delete, open, close, read and write
- **Snapshot** creates a copy of a file or a directory tree at a low cost
- **Append** allows multiple clients to append data to the same file concurrently

Read Operation

1. The application sends a read request
2. GFS client translates the request and sends it to the master
3. Master responds with chunk handle and replica locations
4. The client picks a location and sends a request
5. The chunk server sends the requested data
6. The client forwards the data to the application

Update Order

- **Update (mutation):** An operation that changes the content or metadata of a chunk
- For consistency updates to each chunk must be ordered (Consistency -> replicas have same version of the data)
- To uphold consistency between replicas one replica is designated as the **primary** and the other replicas are **secondaries**
- **Primary** defines the update order

Primary Leases

- For correctness, there is one single primary for each chunk.
- The **Master** is responsible for selecting a chunk server as primary and granting it a lease

- A chunkserver will hold a lease for a period where it will behave as a primary. If the master doesn't hear from the chunk server during a period it will give the lease to another chunk server

Write Operation

1. The application sends a request
2. GFS client translates it and sends it to the master
3. Master responds with chunk handle and replica locations
4. The client pushes write data to all locations where the data then is stored in the chunk server's internal buffer
5. The client sends a write command to the primary
6. The primary determines the order in its buffer and writes it in the order to the chunk
7. The primary sends the serial order to the secondary

Write Consistency

- Primary enforces one update order across all replicas for concurrent writes
- The primary will wait until all replicas are finished writing before sending back a reply
- Doing this will ensure identical replicas, but because the received order of data is different from the replicas there might be mingled fragments and therefore writes are **consistent** but **undefined state** in GFS.

Append Operation

1. Application sends request
2. client translates it and sends it to the master
3. The master responds with a chunk handle and replica locations
4. the client pushes write data to all locations
5. The primary check is if the record fits in a specified chunk
6. If the record does not fit then the primary will:
    a. Pads the chunk
    b. tells secondaries to do the same
    c. informs the client
    d. the client then retries the append with the next chunk
7. If the record fits then the primary will:
    a. Append the record
    b. Tell secondaries to do the same
    c. Receive responses from the secondary
    d. respond to the client

Delete Operation

- Metadata operations

1. Renames a file to a special name
2. After a certain time the actual chunks
3. Supports undelete for a limited time
4. Actual lazy garbage collection

Single Master

- The master has global knowledge of the whole system which also helps with simplified design and hopefully never bottlenecks.

Namespace Management and Locking

- Namespace as a lookup table mapping pathnames to metadata
- Each master requires a set of locks before it runs. Like read locks on internal nodes and read/write locks on the leaf
- **Example**: creating multiple files (f1 and f2) in the same directory (/home/user/).
    - Each operation acquires a read lock on the directory name /home/user/. Prevents the directory from deleting, renaming, or snapshotting. Allows concurrent mutations in the same directory
    - Each operation acquires a write lock on the file names f1 and f2
- **Replica Placement**
    - Maximize data **reliability, availability,** and **bandwidth utilization**
    - The master determines the replica placement
- **Creation, Re-replication and Re-balancing**
    - **Creation**
        - Place a new replica on a chunk server with below-average disk usage
    - **Re-replication**
        - When the number of available replicas falls below the user-specified goal
    - **Re-balancing**
        - Periodically for better disk utilization and load balancing

Garbe Collection

- File deletion logged by master
- The file was renamed to a hidden name with a deletion timestamp
- Master removes hidden files older than 3 days
- Until then hidden files can be read and undeleted
- After deletion, in-memory metadata is also erased

State replica Detection

- If a chunk replica becomes stale meaning that it failed and missed a mutation
- Needs to see the difference between up-to-date and stale replicas
- Uses a chunk version number which is increased when the master grans a new lease on the chunk
- Stale replicas are deleted by the master in regular garbage collection

Fault Tolerance

- Chunk replication
- Data integrity such as checksum which is checked every time an application reads the data
- All chunks are versioned

- The master state is replicated on multiple machines
- When a master fails it can restart almost instantly
- A Shadow master provides only read-only access to the file system when the primary master is down

GFS and HDFS

GFS	HDFS
Master	Namenode
Chunkserver	DataNode
Operation Log	Journal, Edit Log
Chunk	Block
Random file writes possible	Only append is possible
Multiple write/reader model	Single write/multiple reader model
Default chunk size: 64MB	Default chunk size: 128MB

Database and Database Management System

- Database: an organized collection of data
- Database management system: a software to capture and analyze data

Relational SQL Databases

- Dominant technology for storing structured data
- SQL consists of a rich toolset and is easy to use and integrated

ACID

- **Atomicity:** All statements are either executed or the whole transaction is aborted
- **Consistency:** A database is consistent state before and after a transaction
- **Isolation:** Transactions can not see uncommitted changes
- **Durability:** Changes are written to a disk before a database commits transactions so that the committed data is truly written

Challenges

- Web application causes spikes
- RDBMS were not designed to be distributed

NoSQL

- **Avoids:** Overhead of ACID properties, complexity of SQL query
- **Provides:** Scalability, large data volumes, and easy frequent changes

Availability vs Consistency

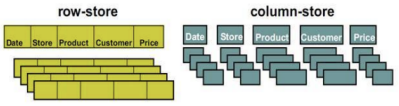
- Availability
    - Replicating data to improve the availability of data
    - Data Replicatiom by storing data in more than one site or node
- Consistency
    - **Strong consistency** means that after an update is completed any subsequent access will return the update value
    - **Eventual consistency** means that it is not guaranteed that subsequent access will return the updated value because of an inconsistency window.
- Any large-scale application has to be reliable: **consistency, availability, partition,** and **tolerance**
- Archive ACID properties on large-scale applications are therefore challenging

CAP Theorem

Out of the three conditions you can only uphold two of them at any given time

- **Consistency:** Consistent state of data after an operation
- **Availability:** Always being able to read and write data
- **Partition Tolerance:** Continue operation in the presence of network partitions

NoSQL Data Models



- Key-Value Data Models
  - Collection of key-value pairs
  - Ordered by Key.Value
  - Example: Dynamo, Scalaris, Voldemort, Riak…
- Column-Oriented Data Model
  - Similar to key/value but the value can have multiple attributes
  - Column is a set of data values of a particular type
  - Store and process data by column instead of by row
  - Example: BigTable, HBase, Cassandra
- Document Data Model
  - Similar to column-oriented but values can have complex documents
  - Is flexible with schemas
  - Example: CouchDB, MongoDB
- Graph Data Model
  - Uses a graph structure with nodes, edges, and properties
  - Example: Neo4J, InfoGrid

BigTable

- Semi-structured data at Google
- Distributed multi-level map
- Provides strong consistency and partition tolerance but not availability

Data Model

- **Table:**
  - Distributed multi-dimensional sparse map
- **Rows:**
  - Every read or write in a row is atomic
  - Rows sorted in lexicographical order
- **Column**
  - The basic unit of data access
  - Column families a groups of the same type of column keys
  - Column key naming: family:qualifier
- **Timestamp:**
  - Each column value may contain multiple versions
- **Tablet:**
  - A contiguous range of rows stored together
  - tablets are split by the system when they become too large
  - Each tablet is served by exactly one tablet server

System Structure

- Master
  - Assign tablets to the tablet server
  - Balances tablet server load

    - Garbage collection
    - Handles schema changes like table and column family creations
- Tablet server
    - Can be added or removed dynamically
    - Each manages a set of tablets (typically 10-100 tablets/server)
    - Handles read/write requests to tablets
    - splits tablets when too large
- Client library
    - Library linked to every client
    - Client data does not move through the master
    - The client communicates directly with tablet servers for reads/writes

Building Blocks

- Google File System
    - Large-scale distributed file system
    - Stores log and data files
- Chubby
    - Ensures there is only one active master
    - Store bootstrap location of BigTable data
    - Discover tablet servers
    - Store BigTable schema and access control lists
- SSTable
    - File format internally to store BigTable data
    - Chunkcs of data plus block index
    - The immutable, sorted file of key-value pairs
    - Each SSTable is stored in a GFS file

Master Startup

The master executes the following steps at startup:

1. Grabs a unique master lock in Chubby to prevent concurrent masters
2. Scans the server directory in Chubby to find the live servers
3. Communicates with every live tablet server to discover
4. Scans the METADATA table to learn the set of tablets

Tablet Assignment

- 1 tablet -> 1 tablet server
- Master uses Chubby to keep track of live tablet servers and unassigned tablets
- Master detects the status of the lock of each tablet server by checking periodically
- Master is responsible for finding when the tablet server is no longer serving its tablets

Finding a Tablet

- **Three level hierarchy**

1. The first level is a file stored in Chubby that contains the location of the root tablet
2. Root tablet contains the location of all tablets in a special METADATA table
3. METADATA table contains the location of each tablet under a row

4. Client library caches tablet locations

Tablet Serving

- Updates committed to a commit log
- Recently committed updates are stored in memory
- Older updates are stored in a sequence of SSTables
- Provides strong consistency when one tablet server is responsible for a given piece of data. Replication is handled on the GFS layer
- The trade-off for this is availability where if a tablet fails its portion of data is temporarily unavailable until a new server is assigned

BigTable vs HBase

BigTable	HBase
GFS	HDFS
Tablet Server	Region Server
SSTable	StoreFile
Memtable	MemStore
Chubby	ZooKeeper

Cassandra

- Column-oriented database
- Created by Facebook
- Provides Availability and partition tolerance but not consistency
- Uses SSTable disk storage for append-only commit log, Memtable for buffering and sorting, and Immutable SSTable files

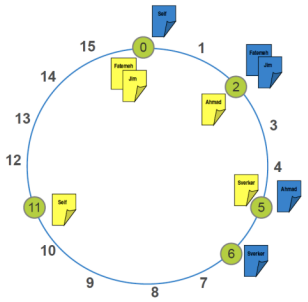
Data Partitioning

- Key/value is stored as objects
- If the size of the data exceeds the capacity of a single machine partitioning happens
- Uses consistent hashing for partitioning. Hashes Both data and node IDs use the same hash function

Consistent hashing good video:
https://www.youtube.com/watch?v=UF9Iqmg94tk

Replication

- To achieve high availability and durability data is replicated on multiple nodes



Adding and Removing Nodes

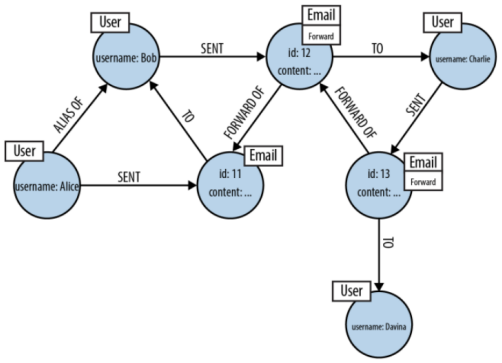
- Gossip-based mechanism where periodically each node contacts another randomly selected node

Neo4j

- Graph database
- Relations between data is important
- Cypher similar query language like SQL
- Provides strong consistency and availability but does not partitioning tolerance

Data Model

- **Node (vertex)**
  - The main data element (Like a specific user information)
- **Relationship (Edge)**
  - May contain direction, metadata like weight or relationship type
- **Label**
  - Define node category (optional)
  - can have more than one
- **Properties**
  - Enrich a node or relationship



4. Scala

https://id2221kth.github.io/slides/2023/04_scala.pdf

Always use immutable variables by default, unless they HAVE to be mutable.

Mutable: Can be changed
Immutable: Cannot be change

5. Parallel Processing

https://id2221kth.github.io/slides/2023/05_parallel_processing_part1.pdf

https://id2221kth.github.io/slides/2023/06_parallel_processing_part2.pdf

- Scale Up also called scale vertically is the process of adding resources to a single node
- Scale out or scale horizontally by adding more nodes to a system

MapReduce

Easy explanation:
https://www.youtube.com/watch?v=cHGaQz0E7AU&ab_channel=ByteMonk

- MapReduce takes care of parallelization, fault tolerance, and data distribution
- Hide system-level details from programmers
- An issue with MapReduce is that it is expensive and slow it always goes to disk and HDFS

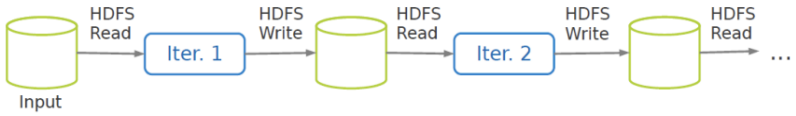


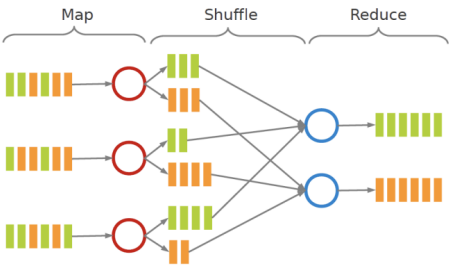
Figure: Displaying the issue with MapReduce

Programming Model

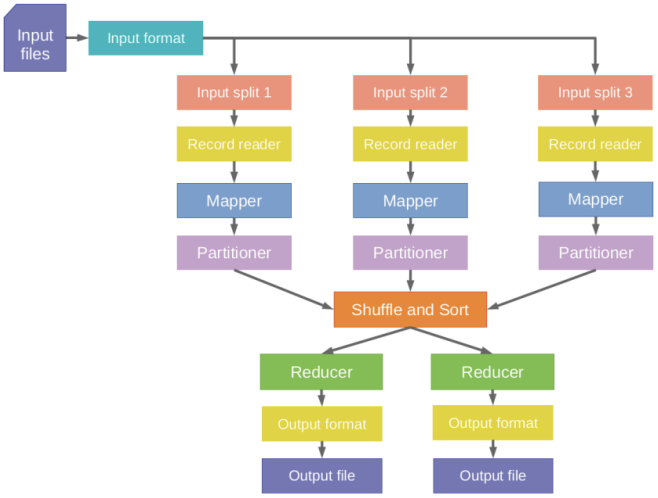
- Model to batch process large data set
- Execution framework to run parallel algorithms
- If a file fits in memory it is directly executed but if not it will utilize data parallel processing by dividing the data and processing them together

Stages

1. Map
    a. Each Map task usually operates on a single HDFS block
    b. Map Task usually runs on the node where the block is stored
    c. Each Map Task Generates a set of intermediate key-value pairs
2. Shuffle and Sort
    a. Sort and consolidate intermediate data from all mappers
3. Reduce
    a. Each Reduce task operates on all intermediate values associated with the same intermediate key

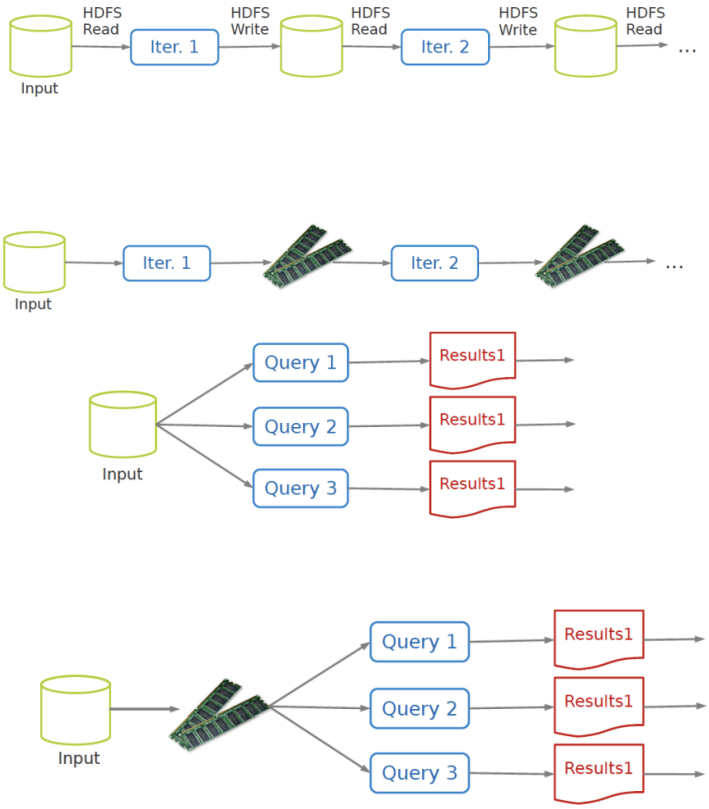


Data Flow



Spark

The difference between Spark and MapReduce is as shown in the images. Spark Utilizes RAM to avoid reading and writing constantly back and forth to the HDFS disks.



Good explanation:
https://www.youtube.com/watch?v=ymtq8yjmD9I&ab_channel=nullQueries

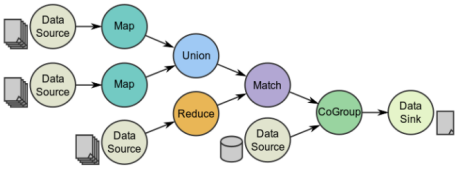
Spark Applications Architecture

Spark application consists of:

- **A Driver process**
    - The heart of a spark application
    - handles the main() function
    - Responsible for: Maintaining information about the application, responding to a user program and input, and analyzing, distributing, and scheduling word across executors
- **A set of executor processes**
    - Executes code assigned to a driver
    - Reports the current state of the computation on the executor to the driver (health)
- **Spark Session**
    - Driver process that controls the spark application
    - A correspondence between a SparkSession and a Spark application
- **Spark Context**
    - The entry point for low-level API functionality
    - Access it through SparkSession

Programming Model

- **Job** described based on a directed acyclic graph (DAG) data flow
- A data flow is composed of a number of data sources, operators, and data sinks by connecting their inputs and their outputs
- Uses parallelizable operators



Resilient Distributed Datasets (RDD)

- A distributed memory abstraction
- Immutable collections of objects spread across a cluster (for example a Linked List)
- An RDD is divided into a number of partitions
- Partitions can be stored on different nodes of a cluster
- The Spark code that is being run is compiled down to an RDD

- There are two types of RDDs. They represent a collection of objects.
    - Generic
    - Key-value. It has special operations such as aggregation and custom partitioning by key
- RDD Operations
    - **Transformations**
        - Allows to build a logical plan
        - Creates a new RDD from an existing one

        - All transformations are lazy meaning that they do not compute their result right away. They are only computing when an action requires a result to be returned to the driver program
        - **Lineage:** transformation used to build an RDD, RDDs are stored as a chain of objects capturing the lineage of each RDD
    - **Actions**
        - Allow us to trigger the computation
        - Run an action to trigger the computation
        - **Three kinds of actions:** view data in the console, collect data to native objects, and write to output data sources
    - Use kvChars.reduceByKey(addFunc). ReduceByKey is better than groupByKey because groupByKey requires all executors to hold all values for a given key in memory before applying while ReduceByKey reduce happens within each partition
    - val kvChars = ... // (t,1), (a,1), (k,1), (e,1), (i,1), (t,1), (e,1), (a,1), (s,1), (y,1), (,,1), ...
    - val redChar = kvChars.reduceByKey(addFunc) // (t,5), (h,1), (,,1), (e,3), (a,3), (i,3), (y,1), (s,4), (k,1))
- **Caching**
    - When an RDD is cached each node stores any partitions it has in memory
    - an RDD that is not cached is re-evaluated each time an action is invoked
    - Two types of caching, cache which saves RDD into memory, and persist(level) which caches to memory on disk
- **Checkpointing**
    - Saves RDD to a disk
    - Checkpoint data is not removed after SparkContext is destroyed

Structured Data Processing

https://id2221kth.github.io/slides/2023/07_structured_data_processing.pdf

DataFrames and DataSets

- Spark has two structured collections; DataFrames and DataSets
- They are distributed collections with well-defined rows and columns.
- Immutable and lazily evaluated plans.

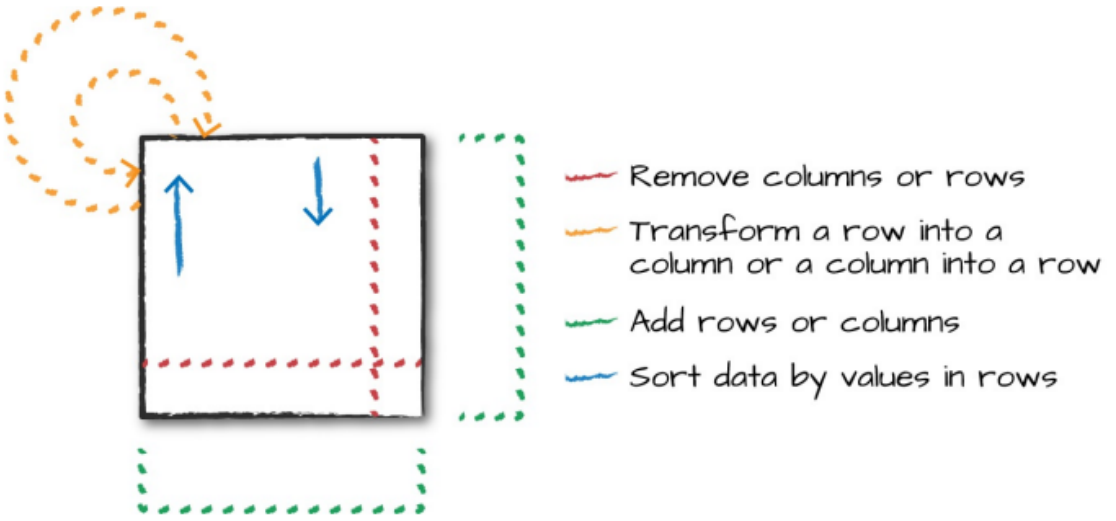
DataFrame

- Consists of a series of rows and a number of columns
- Equivalent to a **table** in a relational database
- Two ways to **create** a DataFrame:
    - From an RDD (Resilient Distributed Dataset)
    - From raw data sources
- A DataFrame is a distributed collection of data organized into named columns.
- It is similar to a table in a relational database or a data frame in R or Python.

- DataFrames are based on the concept of schema, which means that each column has a well-defined data type.
- DataFrames use a schema to allow for optimization in terms of memory and execution speed.
- They provide a high-level, domain-specific API for manipulating data.
- While they are highly optimized, they lack the type safety found in Datasets.

Dataframe Transformations

- Add and remove rows or columns
- Transform a row into a column (or vice versa)
- Change the order of rows based on the values in columns



[M. Zaharia et al., Spark: The Definitive Guide, O'Reilly Media, 2018]

Dataframe Actions

Like RDDs, DataFrames also have their own set of actions.

- **collect**: returns an array that contains all of rows in this DataFrame.
- **count**: returns the number of rows in this DataFrame.
- **first** and **head**: returns the first row of the DataFrame.
- **show**: displays the top 20 rows of the DataFrame in a tabular form.
- **take**: returns the first n rows of the DataFrame.

Aggregation

Overview of aggregation in Spark SQL. In an aggregation you specify:

- A key or grouping
- An aggregation function

The given function must produce one result for each group.

Joins

- Joins are relational constructs you use to combine relations together.
- There are different types of join.
- Two main communication ways during joins
    - **Shuffle join:** big table to big table
    - **Broadcast join**: big table to small table

DataSet

- A **DataFrame** is a distributed collection of data organized into named columns. A **DataSet** is a distributed collection of data with the benefits of **both DataFrames and RDDs** (Resilient Distributed Datasets).
- DataSet API unifies the DataFrame and RDD APIs.
- DataSets are strongly typed, thus providing type safety, making them more suitable for use cases where you need to work with a specific data type.
- They can hold both structured and unstructured data, and you can use native programming language features for type-specific operations.
- DataSets have the ability to optimize the execution plan and generate more efficient code due to their type-specific nature.
- They provide a combination of functional programming and SQL-like operations.
- Datasets are essentially a type-safe version of DataFrames.

Remember: RDD vs DataFrame vs DataSet

Example DataFrame DataSet

FROM HOMEWORK 3, QUESTION 1:

“1. Briefly compare the DataFrame and DataSet in SparkSQL and via one example show when it is beneficial to use DataSet instead of DataFrame.
DataFrames and DataSets are both part of the Spark SQL model as high-level abstractions for working with structured (and unstructured?) data. They are both distributed table-like collections with well-defined rows and columns

Just like Resilient Distributed Datasets (RDDs), DataFrames are distributed collections of data. DataFrames are however organized into a series of rows and a number of named columns, similar to a table in a relational database. DataSets are an even newer addition to Spark and can be seen as an extension to DataFrames, designed to combine the strong typing of RDDs with the optimization benefits of DataFrames.

DataFrames does not provide compile-time type safety, meaning it does not throw an error during the compile time, only when the code is being executed. This differs from DataSets,

which throws the error during the compile time. DataSets also provides advanced encoders, which can provide on-demand access to individual attributes. This feature is not found in DataFrames.”

Stream Processing

https://id2221kth.github.io/slides/2023/08_stream_processing_part1.pdf
https://id2221kth.github.io/slides/2023/09_stream_processing_part2.pdf

Stream processing refers to the continuous handling of data as it arrives in a continuous flow. In stream processing, input data is **unbounded**, meaning it is a continuous event with no predetermined beginning or end. Often used for processing credit card transactions, website clicks etc.

Database Management Systems (DBMS) perform **data-at-rest** analytics by storing and indexing data before processing. Stream Processing Systems (SPS) instead perform **data-in-motion** analytics, processing the information continuously as it flows without persistently storing it.

Streaming Data

- Data Streams: Unbounded data broken into a sequence of individual tuples.
- Data Tuple: the atomic data item in a data stream. Can be structured, semi-structured, or unstructured.

Streaming Processing Patterns

Two common patterns for stream processing are **micro batch systems** and **continuous processing-based system**s.

- Micro Batch Systems: Slices unbounded data into bounded data sets and processes them in batcehs
- Continuous Processing-Based Systems: Nodes continually listen to messages and output updates

Event and Processing Time:

- Event Time: The time at which events actually occur
- Processing Time: The time when the records are received at the streaming application

Ideally, the event and processing times should be equal, but there can be a skew between them.

Windowing

A window is a buffer that is associated with an input port to retain preciously received tuples. There are different windowing management policies, such as count-based policies and time-based policies. Windows can also be of two types: tumbling and sliding.

- Tumbling: Supports batch operations
- Sliding: Supports incremental operations

Triggering and Windowing

Triggering determines **when** in **processing time** the results of grouping are emitted as panes, while windowing determines **where** in **event time** data is grouped for processing.

Time Based Triggering (Processing Time) & Windowing (Event Time)

Time-based Triggering: In this method, the system buffers incoming data into windows until a certain amount of processing time has passed.
Time-based Windowing: Reflects the times at which events ACTUALLY happened, handles out-of-order events using watermarking to measure progress in event time.

Partitioned logs

Typical message brokers delete messages once consumed, but log-based message brokers will durably store all events in a sequential log. **Producers append** messages and the **consumers sequentially read** from the log.

Kafka - A Log-Based Message Broker:

What is Kafka? It is a distributed, topic-oriented, partitioned, replicated commit log service. Kafka provides topics, partitions, and ordered, append-only logs for messaging.

Delivery Guarantees

Kafka guarantees thaht messages from a single partition are delivered to a consumer in order. There is however **no** guarantee on the ordering of messages from **different partitions**. Kafka guarantees **at-least-once delivery**.

Spark Streaming

https://id2221kth.github.io/slides/2023/09_stream_processing_part2.pdf

These slides introduce the concept of Structured Streaming, which is related to Spark Streaming.

Structured Streaming

A high-level stream processing engine in Apache Spark. It treats a live data stream as a table that is continuously appended to.

- It defines a query on the input table as a static table. Spark automaticall converts this batch-like query into a streaming execution plan.
- You can specify triggers to control when to update the results. Each time a trigger fires, Spark checks for new data and incrementally updates the result.

There are three output modes:

- Append: New rows appended since the last trigger
- Complete: Entire updated result
- Update: Only updated rows since last trigger

Spark Streaming vs Structured streaming

Spark Streaming is the older, micro-batch-based streaming processing engine provided by Apache Spark.

SPARK STREAMING

- Processes data in small, discrete batches. Makes it suitable for application that can tolerate some delay in data processing (a few seconds).
- Works with DStreams, which are essentially a series of RDDs.
- Follows a discrete, batch processing model where you process data as it arrives but it is done in small time intervals.

STRUCTURED STREAMING

- Newer, high-level API for stream processing introduced in Apache spark.
- Treats live data stream as a continuous, unbounded table. Does not use micro-batches, but processes data as it arrives, just like traditional batch processing.
- Structured Streaming is designed to provide a more consistent and simpler API for both batch and stream processing. Built on top of Spark SQL, which is a structured and expressive API for querying structured data.
- Allows you to write streaming queries as if you were working with static, structured data. Makes stream processing logic easier.

Large Scale Graph Processing

https://id2221kth.github.io/slides/2023/10_graph_processing.pdf

Lecture involves handling and analyzing graphs with a massive number of nodes and edges.