ADVANCED TOPICS IN BUSINESS INTELLIGENCE

UNIT 5

CONTENT

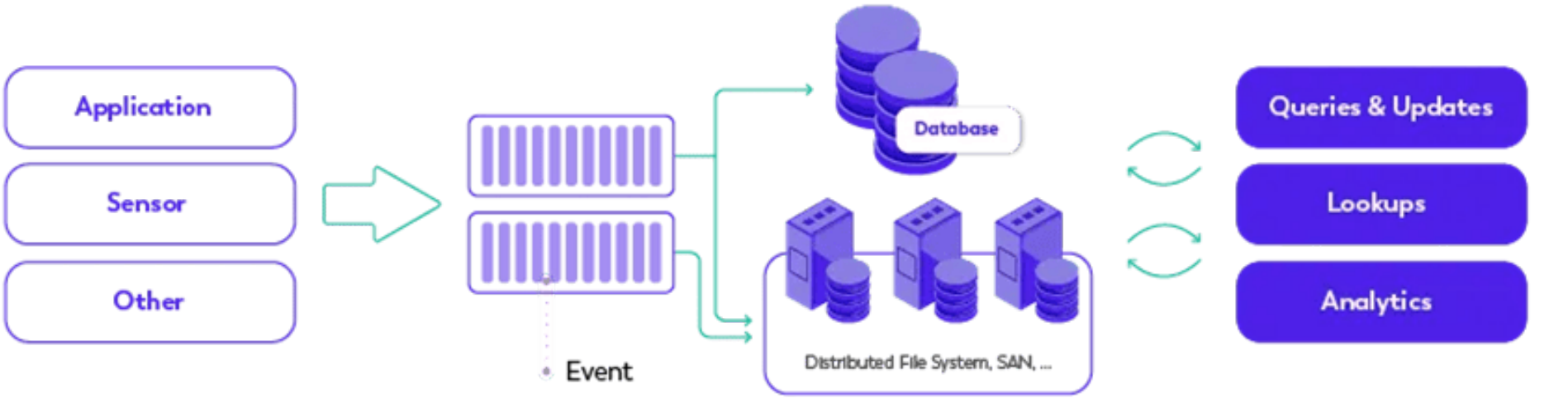
Real-time analytics and stream processing

Big data analytics

- Hadoop
- Spark
- NoSQL databases

Prescriptive analytics and optimization techniques

Emerging trends and future directions in business intelligence and analytics



Birth of the data
as streams of events

Storing data
at rest

Applications schedule
computation on the data

STREAM PROCESSING

Real-time stream processing

the continuous processing of data as it is generated, enabling immediate analysis and decision-making.

Instead of waiting for data to accumulate and then processing it in batches (as with traditional batch processing),

stream processing handles events as they arrive, providing insights, responses, or transformations instantly or within milliseconds to seconds.

Key Characteristics

Continuous Data Flow:

Data is processed as soon as it arrives, typically in an unbounded and continuous stream.

It’s ideal for environments where data is generated constantly, such as sensor networks, financial markets, or social media feeds.

Low Latency:

there is minimal delay between data ingestion and the processing of that data.

This allows for immediate insights and actions.

Event-Driven:

Data in real-time stream processing is typically in the form of discrete events, such as user clicks, financial transactions, or sensor readings.

Systems respond to these events almost instantly, triggering actions or updating states.

Stateful Processing:

Stream processors often maintain a state across different events.

counting occurrences of events, tracking patterns over time, or joining streams of data.

A system like **Apache Flink** is highly optimized for managing state efficiently in real-

Windowing:

In stream processing, data is often aggregated over windows of time (e.g., every 5 minutes) or a certain number of events (e. g., after every 100 events).

This allows for meaningful analysis and summarization of the continuous data flow.

Use Cases of Real-Time Stream Processing

Fraud Detection:	real-time stream processing can detect suspicious transactions instantly and flag or block them before harm is done.
Real-Time Analytics:	For online platforms, clickstream analysis in real-time helps provide instant insights into user behavior, enabling dynamic content adjustments or recommendations.
Internet of Things (IoT):	Devices and sensors continuously generate data. Real-time processing allows for immediate actions, such as shutting down a machine in case of overheating, or predictive maintenance based on continuous monitoring of equipment.
Financial Market Analysis:	Stream processing is critical in stock exchanges, where real-time data processing is necessary to monitor market movements and execute trades based on live data.
Real-Time Recommendations:	Services like Netflix and Spotify use real-time stream processing to analyze user interactions and provide personalized recommendations immediately.

Technologies Used for Real-Time Stream Processing

Apache Flink:	Apache Kafka:	Apache Storm:	Apache Spark Structured Streaming:
• Optimized for real-time, stateful stream processing with low-latency and fault tolerance features.	• Often used as a distributed message broker, handling data streaming between producers and consumers.	• Another real-time stream processing framework, designed for distributed and fault-tolerant processing.	• A micro-batch processing framework for near real-time analytics, which offers a simpler approach to streaming.

Telecom Churn Prediction

Scenario:	A telecom company wants to predict customer churn in real-time to reduce attrition. They implement a real-time churn prediction system based on customer usage patterns and interactions with support.
Data Source:	Stream customer call logs, data usage, service complaints, and billing information through **Kafka**.
Flink for Stream Processing:	Use **Flink** to preprocess the data, calculate real-time metrics (e.g., call duration, internet usage over the past 24 hours), and detect patterns (e.g., customers frequently contacting support). Real-time features such as "days since last payment" or "increase in service complaints" are computed.
Churn Prediction Model:	A pre-trained **logistic regression** model is integrated into the Flink pipeline to predict the probability of churn.
Real-Time Action:	If the churn probability is high, the system triggers an SMS offering a discount or enhanced service. If the customer is still dissatisfied, it escalates to a human support team for intervention.

Churn Prediction

Churn prediction is the process of identifying customers who are likely to stop using a service or product.

In real-time, this involves processing and analyzing customer data as it is generated, allowing companies to intervene promptly and reduce customer attrition.

Real-time churn prediction enables businesses, such as telecom companies, online platforms, and subscription services, to take immediate action, such as offering incentives or sending notifications to at-risk customers.

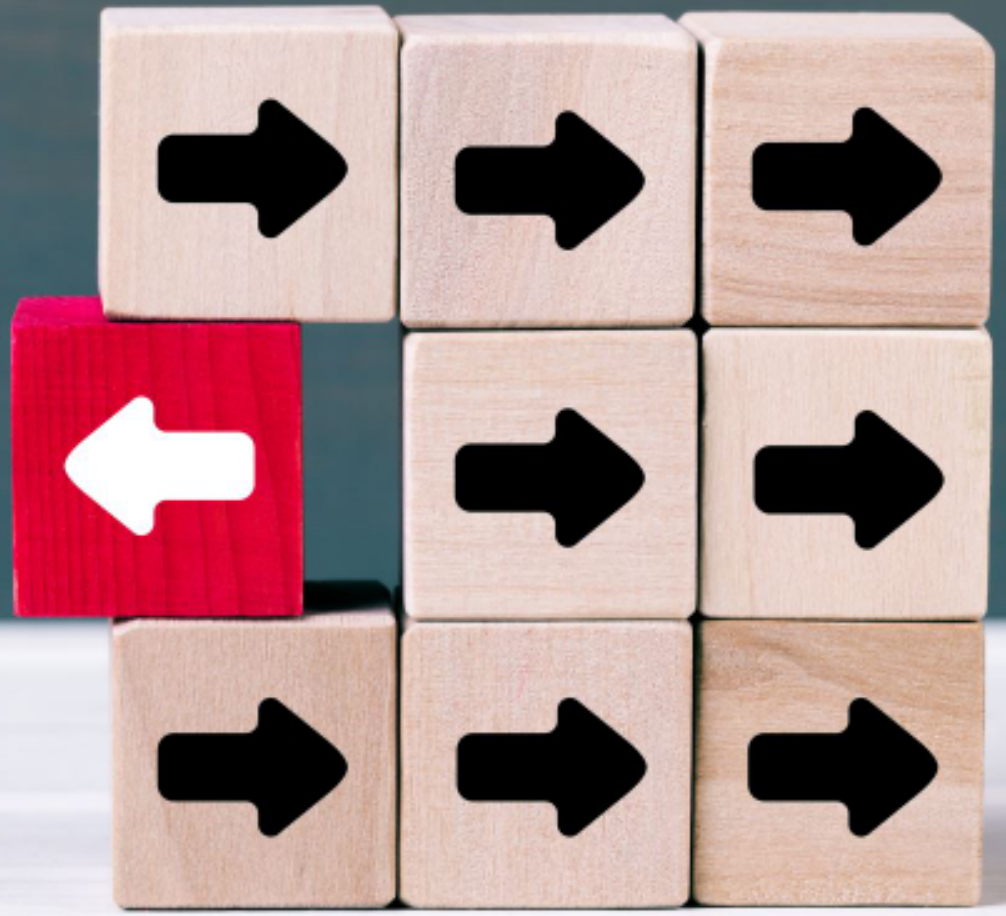
Telecom company wants to predict customer churn

The telecom company wants to prevent churn by analyzing real-time customer data, such as:

Call durations.
Data usage.
Interaction with customer support.
Billing and payment history.
Network issues or service complaints.

The goal is to use this data to predict which customers are likely to stop using the service soon and intervene with offers, discounts, or personalized communication before they leave.

Step-by-Step Real-Time Churn Prediction Process



Data Collection (Real-Time Ingestion)

Data Sources: The telecom company continuously collects real-time data from multiple sources:

Call logs (e.g., dropped calls, call durations).	**Data usage patterns** (e.g., internet usage, session time).	**Support interactions** (e.g., complaints, service requests).	**Billing information** (e.g., late payments, subscription renewals).

Tool: **Apache Kafka** is used to stream this real-time data into the processing pipeline.

Example: A customer calls telecom support several times in a short period due to network issues, raising the probability of churn.

Preprocessing and Feature Engineering

Data from Kafka is consumed by **Apache Flink** for real-time processing.

Flink cleans and transforms the data and generates relevant features, such as:

Time since last call.	**Number of dropped calls** in the last week.	**Total data consumption** in the last month.	**Frequency of customer support interactions**.	**Payment delays** or missed payments.

Example: A feature like "increased support calls in the last 7 days" or "dropped calls in the last 24 hours" is calculated for every customer.

Churn Prediction Model (Real-Time Scoring)

Machine Learning Model:

- A pre-trained machine learning model (e.g., logistic regression, random forest, or deep learning) is used to predict the churn probability.
- This model is trained on historical data of customer behavior, past churn events, and retention outcomes.

Real-Time Model Scoring:

- As data is processed in real-time, the features generated by Flink are passed to the model to predict whether a customer is likely to churn.
  - **Tools**: The model can be deployed using **Flink ML** or served using **TensorFlow Serving** for real-time predictions.

Example:

- The system computes a churn probability of 85% for a customer who has reported multiple network issues, decreased their data usage, and had delayed payments in the last month.

Real-Time Actions and Interventions

Intervention Strategy: If the churn probability exceeds a threshold (e.g., 80%), the system triggers an action in real-time:

Sending personalized offers (e.g., discounts, data packages).	Escalating to customer service for a follow-up call.	Sending retention emails or SMS messages.

Tools: Integration with CRM systems, email/SMS services, or automated chatbots for direct communication.

Example: If the model predicts that a customer is at high risk of churning, the system might immediately send an SMS offering a 10% discount on the next month's bill or provide additional data usage for free.

Continuous Monitoring and Feedback Loop

Monitoring Customer Behavior:

- After the intervention, customer behavior is monitored in real-time to determine whether the churn prediction was accurate or if the intervention was effective.
  - If the customer continues to use the service or accepts the offer, this positive outcome is tracked.
  - If the customer still shows signs of churn, further actions may be taken, such as direct outreach by the customer support team.

Feedback and Model Retraining:

- The outcomes of interventions (e.g., customer retention or churn) are fed back into the machine learning pipeline to improve the model.
- Periodically, the model is retrained using updated data to reflect the latest customer behavior trends.

Example:

- If the customer accepts the discount offer and continues using the service, the system tracks this behavior and adjusts the churn model to recognize the effectiveness of the discount in preventing churn.

Real-Time Churn Prediction Pipeline (Using Apache Kafka and Apache Flink)

Data Stream Ingestion:	Kafka streams customer activity data in real-time to the processing pipeline.
Data Preprocessing and Feature Engineering:	Apache Flink consumes the Kafka stream, applies transformations, cleans data, and computes real-time features (e.g., "time since last purchase").
Model Scoring:	The real-time features are passed to a pre-trained model (deployed using TensorFlow Serving or Flink ML), which predicts the churn probability.
Real-Time Intervention:	If the churn probability exceeds a certain threshold (e.g., 80%), the system triggers immediate actions, such as sending offers through a CRM system or escalating customer support interactions.
Continuous Monitoring and Feedback:	The pipeline is monitored continuously, and feedback (e.g., whether the customer accepted the offer or stayed) is used to retrain the model periodically.

Customer Lifetime Value (CLV) Estimation

Customer Lifetime Value (CLV) is a metric that estimates the total revenue a business can reasonably expect from a customer throughout their relationship.

It helps companies understand the long-term value of their customers, guiding decisions on marketing, sales, and customer retention strategies.

Accurate CLV estimation allows businesses to optimize customer acquisition costs and focus on retaining the most valuable customers.

Components of CLV

Average Purchase Value (APV):	The average amount a customer spends on a purchase.
Purchase Frequency (PF):	The number of purchases a customer makes over a specific period.
Customer Lifespan (CL):	The length of time a customer continues to make purchases from the business.
Gross Margin (GM):	The percentage of revenue retained after deducting the cost of goods sold (COGS).

CLV Formula

CLV can be estimated using the formula:

$$CLV = APV \times PF \times CL \times GM$$

However, **real-time CLV** estimation considers dynamic customer behavior, and data is constantly updated as new customer interactions occur.

E-Commerce CLV Estimation

Scenario:

- A large e-commerce company wants to estimate the real-time CLV of its customers to personalize marketing strategies and optimize spending on customer acquisition.

Data Sources:

- The company collects real-time data from its website, including transaction history, browsing behavior, and customer interactions with promotional emails.

Data Processing:

- Using **Apache Kafka** and **Flink**, the company streams customer transaction data and calculates real-time metrics like purchase frequency and average order value.

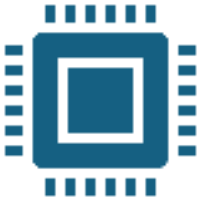
E-Commerce CLV Estimation

CLV Model:

A pre-trained regression model is deployed using **TensorFlow Serving**. The model estimates CLV based on customer purchase history, browsing patterns, and responses to promotions.

Personalized Marketing:

Customers with high CLV are offered loyalty rewards or special discounts, while those with lower CLV receive personalized re-engagement offers via email or SMS.



Real-Time Adjustment:

The CLV model continuously updates based on new customer data, allowing the company to adjust its marketing strategies dynamically.

Benefits of Real-Time CLV Estimation

1. **Personalized Marketing**: Businesses can tailor marketing campaigns to individual customers, maximizing revenue potential.
2. **Improved Resource Allocation**: By understanding which customers are most valuable, companies can focus resources on retention and growth for those segments.
3. **Optimized Acquisition**: Companies can adjust customer acquisition strategies, focusing on acquiring high-CLV customers.
4. **Increased Retention**: Personalized offers based on CLV predictions help retain valuable customers and reduce churn.
5. **Business Growth**: CLV estimation improves decision-making, enhancing profitability and long-term business growth.

Apache Flink

What is Apache Flink?

open-source, distributed engine for stateful processing over unbounded (streams) and bounded (batches) data sets.

Stream processing applications are designed to run continuously, with minimal downtime, and process data as it is ingested.

Apache Flink is designed for low latency processing, performing computations in-memory, for high availability, removing single point of failures, and to scale horizontally.

Apache Flink's features include advanced state management with exactly-once consistency guarantees, event-time processing semantics with sophisticated out-of-order and late data handling.

Apache Flink has been developed for streaming-first, and offers a unified programming interface for both stream and batch processing.

Use of Apache Fink

Event-driven applications

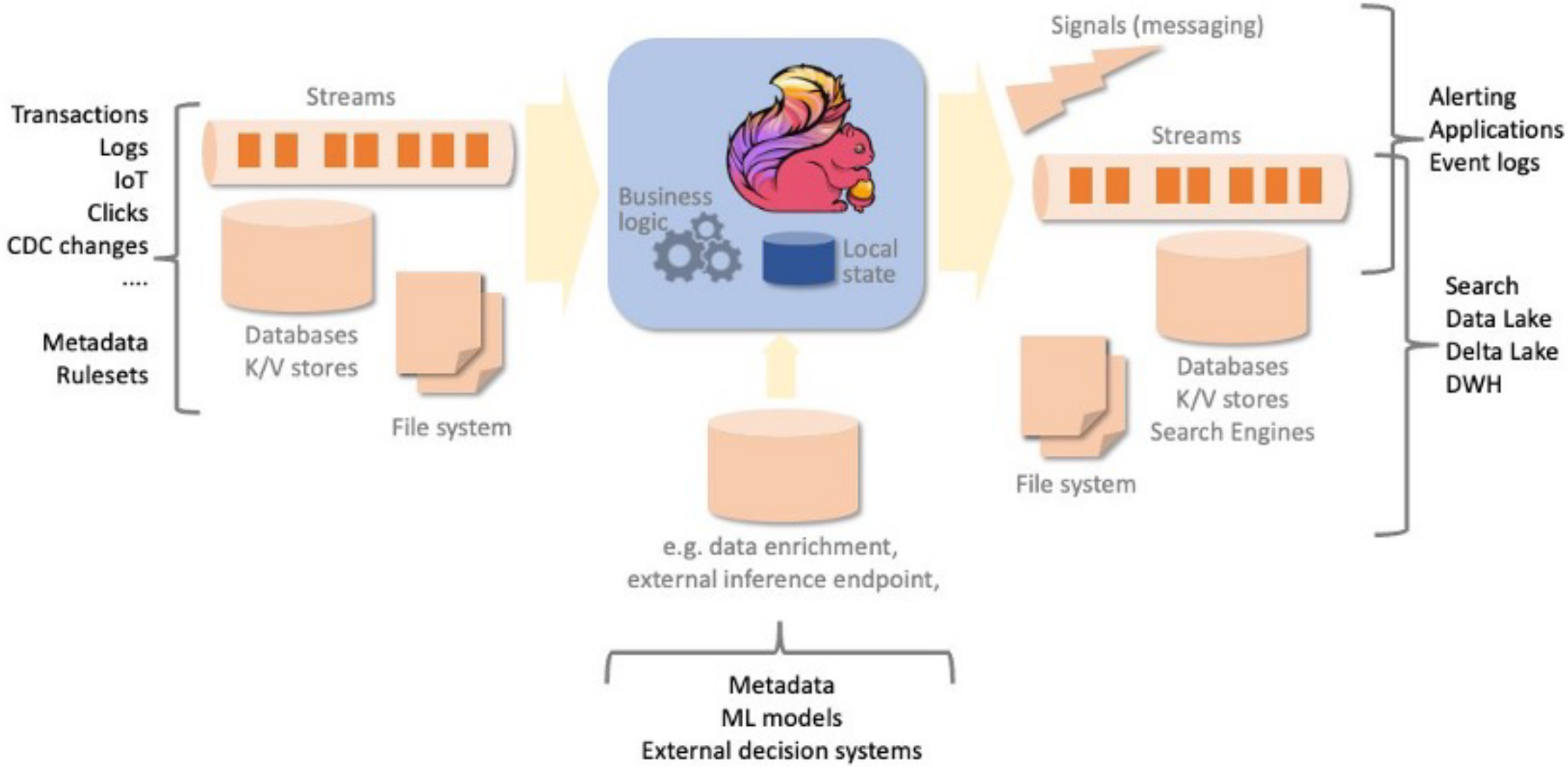
- ingesting events from one or more event streams and executing computations, state updates or external actions.
- Stateful processing allows implementing logic beyond the Single Message Transformation, where the results depend on the history of ingested events.

Data Analytics applications

- extracting information and insights from data.
- Traditionally executed by querying finite data sets, and re-running the queries or amending the results to incorporate new data.
- With Apache Flink, the analysis can be executed by continuously updating, streaming queries or processing ingested events in real-time, continuously emitting and updating the results.

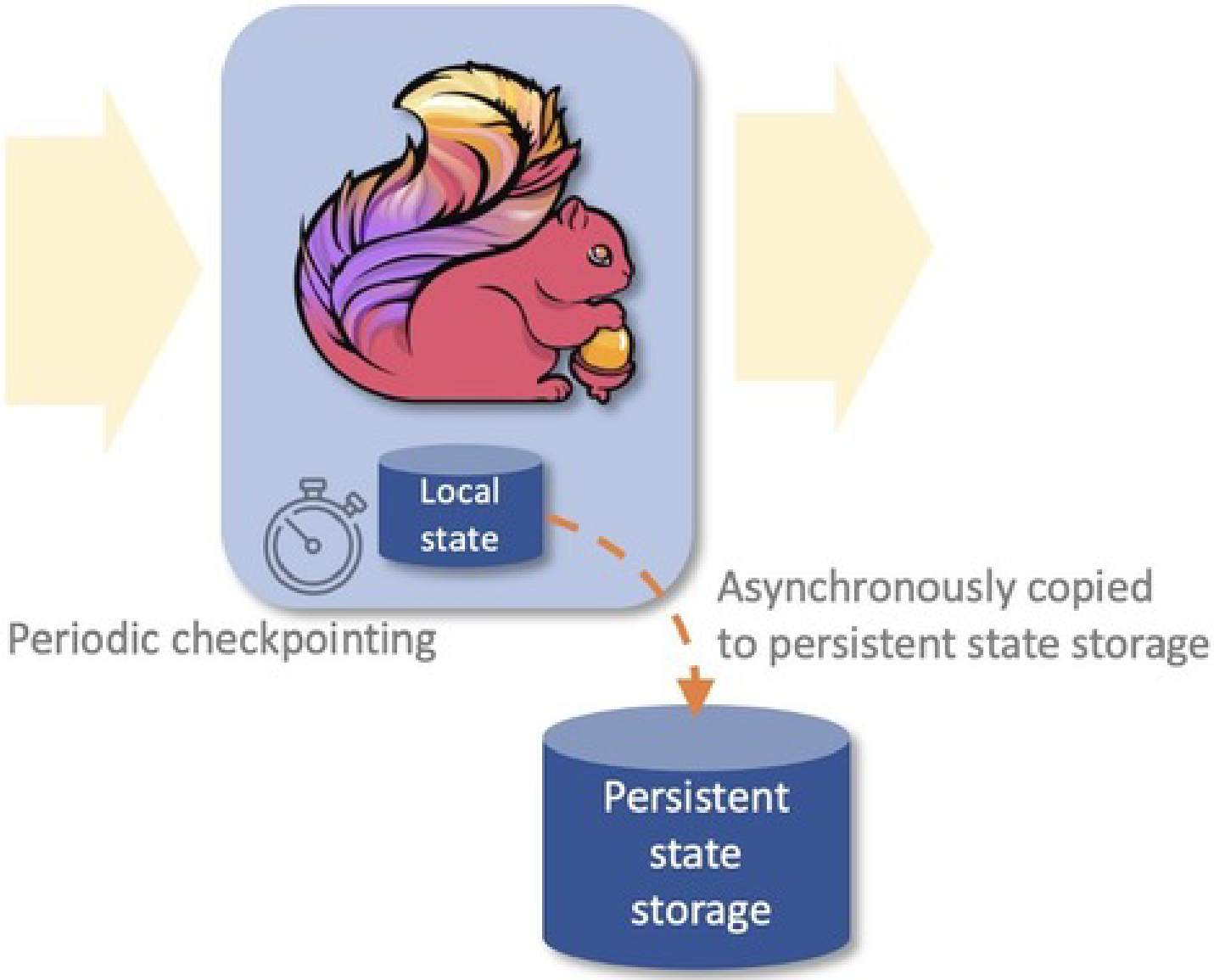
Data pipelines applications

- transforming and enriching data to be moved from one data storage to another.
- Traditionally, extract-transform-load (ETL) is executed periodically, in batches.
- With Apache Flink, the process can operate continuously, moving the data with low latency to their destination.


Transactions
Logs
IoT
Clicks
CDC changes
....
Metadata
Rulesets
Streams
Databases
K/V stores
File system
Business
logic
Local
state
e.g. data enrichment,
external inference endpoint,
Metadata
ML models
External decision systems
Signals (messaging)
Streams
Databases
K/V stores
Search Engines
File system
Alerting
Applications
Event logs
Search
Data Lake
Delta Lake
DWH

How does Apache Flink work?

- Flink is a high throughput, low latency stream processing engine.
- A Flink application consists of an arbitrary complex acyclic dataflow graph, composed of streams and transformations.
- Data is ingested from one or more data sources and sent to one or more destinations.
- Source and destination systems can be streams, message queues, or datastores, and include files, popular database and search engines.
- Transformations can be stateful, like aggregations over time windows or complex pattern detection.
- Fault tolerance is achieved by two separate mechanisms:
  - automatic and periodic checkpointing of the application state, copied to a persistent storage,
  - to allow automatic recovery in case of failure;
  - on-demand savepoints, saving a consistent image of the execution state, to allow stop-and-resume, update or fork your Flink job, retaining the application state across stops and restarts.
  - Checkpoint and savepoint mechanisms are asynchronous, taking a consistent snapshot of the state without “stopping the world”, while the application keeps processing events.


Local state
Periodic checkpointing
Asynchronously copied to persistent state storage
Persistent state storage

What are the benefits of Apache Flink?

Process both unbounded (streams) and bounded (batches) data sets

- Apache Flink can process both unbounded and bounded data sets, i.e., streams and batch data.
- Unbounded streams have a start but are virtually infinite and never end.
- Processing can theoretically never stop.
- Bounded data, like tables, are finite and can be processed from the beginning to the end in a finite time.
- Apache Flink provides algorithms and data structures to support both bounded and unbounded processing through the same programming interface.
- Applications processing bounded data will end their execution when reaching the end of the input data sets.

Run applications at scale

- Apache Flink is designed to run stateful applications at virtually any scale. Processing is parallelized to thousands of tasks, distributed multiple machines, concurrently.
- State is also partitioned and distributed horizontally, allowing to maintain several terabytes across multiple machines.
- State is checkpointed to a persistent storage incrementally.

What are the benefits of Apache Flink?

In-memory performance

- Data flowing through the application and state are partitioned across multiple machines.
- Hence, computation can be completed by accessing local data, often in-memory.

Exactly-once state consistency

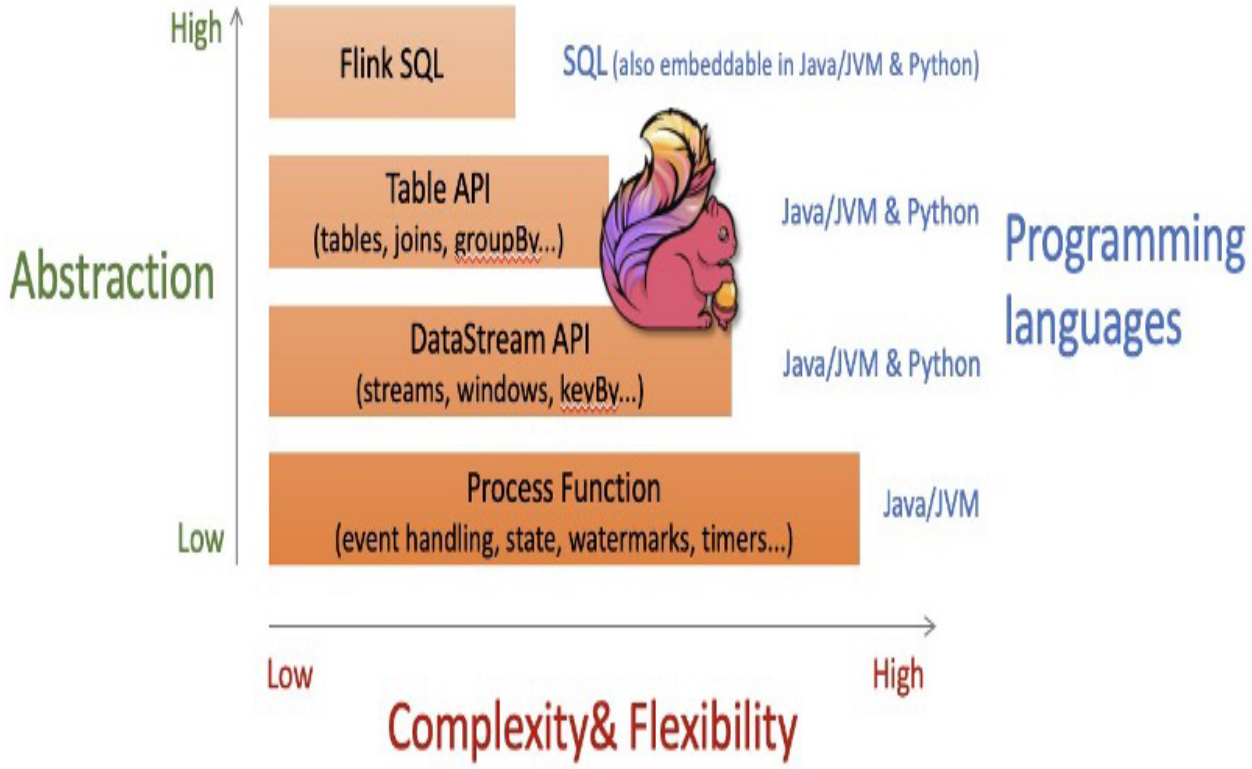
- Applications beyond single message transformations are stateful.
- The business logic needs to remember events or intermediate results.
- Apache Flink guarantees consistency of the internal state, even in case of failure and across application stop and restart.
- The effect of each message on the internal state is always applied exactly-once, regardless the application may

Wide range of connectors

- Apache Flink has a number of proven connectors to popular messaging and streaming systems, data stores, search engines, and file system.
- Some examples are Apache Kafka, Amazon Kinesis Data Streams, Amazon SQS, Active MQ, Rabbit MQ, NiFi, OpenSearch and ElasticSearch, DynamoDB, HBase, and any database providing JDBC client.


Apache Kafka
APACHE hadoop
elasticsearch
RabbitMQ
Amazon DynamoDB
Amazon S3
PULSAR
Apache Cassandra
Amazon Kinesis Data Streams
APACHE nifi

Multiple levels of abstractions



- Apache Flink offers multiple level of abstraction for the programming interface.
- From higher level streaming SQL and Table API, using familiar abstractions like table, joins and group by.
- The DataStream API offers a lower level of abstraction but also more control, with the semantics of streams, windowing and mapping.
- And finally, the ProcessFunction API offers fine control on the processing of each message and direct control of the state.
- All programming interfaces work seamlessly with both unbounded (streams) and bounded (tables) date sets.
- Different levels of abstractions can be used in the same application, as the right tool to solve each problem.


NHL


SmartThings


AUTODESK.


NortonLifeLock™


BT


POSHMARK

Key Technologies and Tools for Big Data Analytics

Key Considerations for Choosing Big Data Tools

Scalability: The ability to handle increasing data volume and complexity.

Flexibility: Compatibility with various data types and sources.

Ease of Use: User-friendly interfaces and strong community support.

Integration: Seamless integration with existing systems and tools.

Cost: Balancing between the tool's capabilities and the budget.

Performance: Efficient processing and real-time data analytics capabilities.

Security: Robust security features to protect sensitive data.

Hadoop Ecosystem

Hadoop Distributed File System (HDFS): A scalable and reliable storage system for large datasets.

MapReduce: A programming model for processing large datasets with a distributed algorithm on a Hadoop cluster.

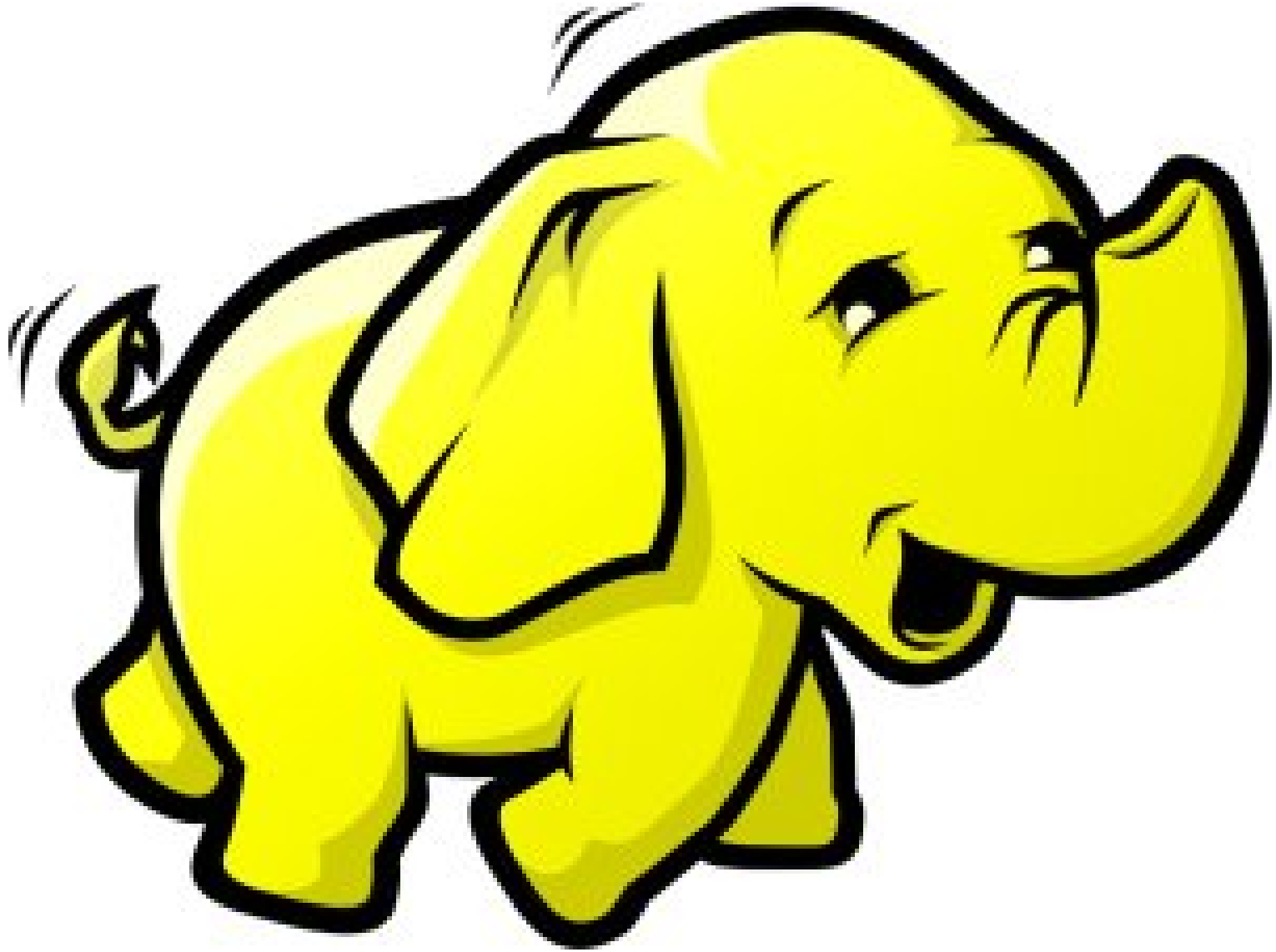
YARN (Yet Another Resource Negotiator): Manages resources in Hadoop and provides a framework for job scheduling and cluster resource management.

HBase: A distributed, scalable, big data store, modeled after Google's Bigtable.

Hive: A data warehouse infrastructure built on top of Hadoop for providing data summarization, query, and analysis.

Pig: A high-level platform for creating MapReduce programs used with Hadoop.

Say hello to Hadoop



HADOOP

- Hadoop is an open-source framework used for storing and processing large datasets in a distributed computing environment. It is designed to scale up from a single server to thousands of machines, each offering local computation and storage.

What's all the excitement about?

Big Data

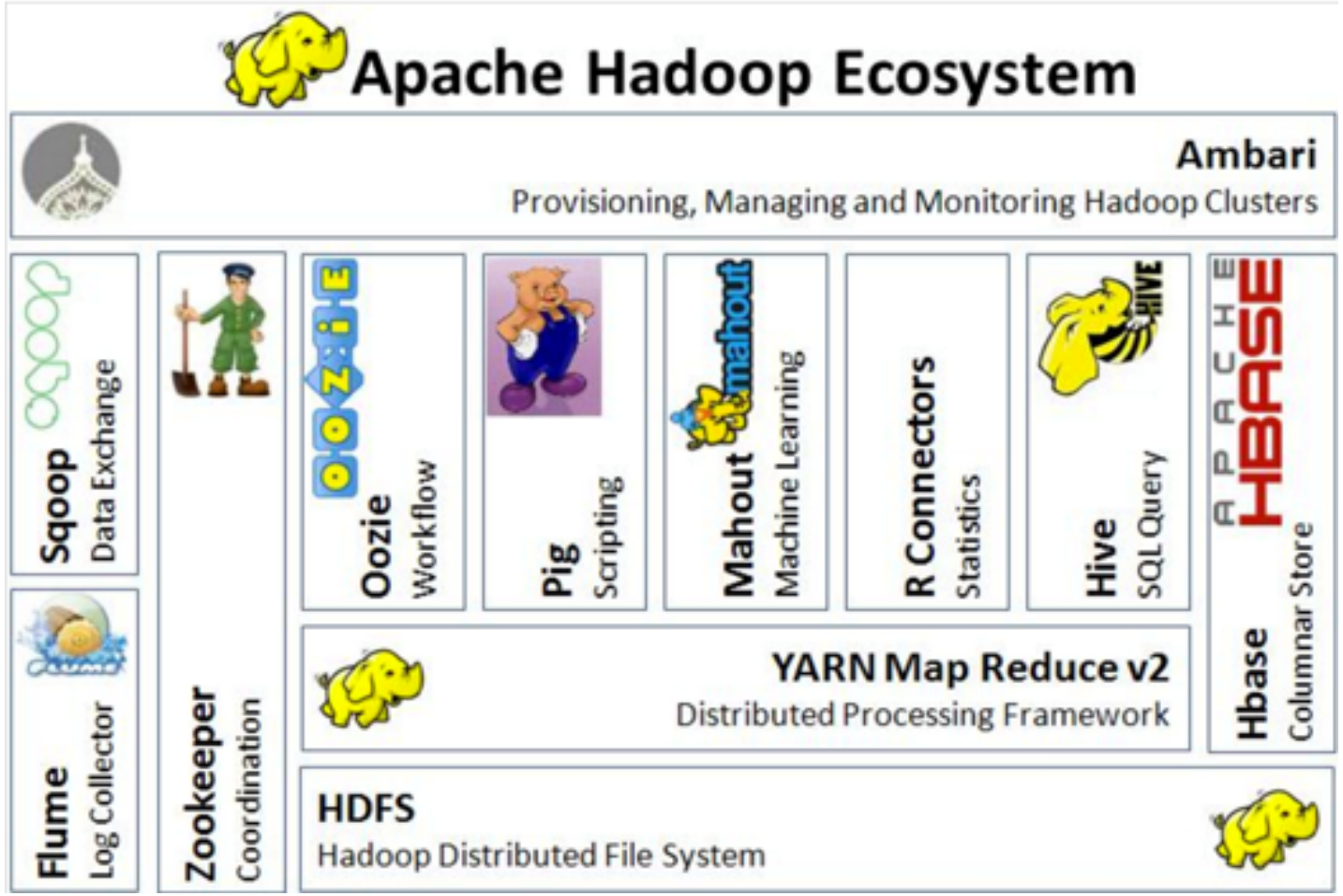
Analytics

- Data science: Develop insights from Big Data
  - Often statistical, often centered around Clustering and Machine Learning
- Big Brother Is Watching You (BBIWY)
  - Governments? Corporations? Criminals?
  - Somebody else's presentation topic!
    - Bruce Schneier, et al

Cloud Computing

- Public: Amazon, et al, low-cost prototyping for startup companies
  - Elastic Compute Cloud (EC3)
  - Simple Storage Service (S3)
  - EBS
  - Hadoop services
- Private

Grid or Cluster computing


Apache Hadoop Ecosystem
Ambari
Provisioning, Managing and Monitoring Hadoop Clusters
Sqoop
Data Exchange
Flume
Log Collector
Zookeeper
Coordination
Oozie
Workflow
Pig
Scripting
Mahout
Machine Learning
R Connectors
Statistics
Hive
SQL Query
APACHE
Hbase
HBASE
Columnar Store
YARN Map Reduce v2
Distributed Processing Framework
HDFS
Hadoop Distributed File System

Hadoop: The High Level

Apache top-level project

- develops open-source software for reliable, scalable, distributed computing.”

Software library

- framework that allows for distributed processing of large data sets across clusters of computers using a simple programming model
- designed to scale up from single servers to thousands of machines, each offering local computation and storage
- designed to detect and handle failures at the application layer…
- delivering a highly-available service on top of a cluster of computers, each of which may be prone to failures.”

Hadoop Distributed File System (HDFS)

- primary storage system used by Hadoop applications.
- HDFS creates multiple replicas of data blocks and distributes them on compute nodes throughout a cluster to enable reliable, extremely rapid computations.

MapReduce

- a programming model and software framework for writing applications that rapidly process vast amounts of data in parallel on large clusters of compute nodes.

What's Hadoop Used For?
Major Use Cases per Cloudera

Cloudera, a company providing enterprise data cloud solutions, identifies several key use cases for Hadoop, including:

Data Warehousing:

Organizations use Hadoop to store and analyze large volumes of structured and unstructured data.

Data Lake:

Hadoop can act as a centralized repository for storing raw data, which can be processed and analyzed later as needed.

Batch Processing:

Hadoop excels in processing large datasets in batches, which is useful for tasks like log processing, ETL (Extract, Transform, Load) operations, and more.

Real-time Processing:

While traditionally used for batch processing, Hadoop components like Apache Storm and Spark enable real-time data processing and analytics.

Machine Learning:

Hadoop's scalable nature makes it a suitable platform for machine learning tasks, including training models on large datasets.

Data Archiving:

Hadoop is also used for long-term data storage, ensuring that historical data is preserved and can be accessed when needed.

HDFS

Google File System (GFS) as the Original Inspiration:

The Hadoop Distributed File System (HDFS) was inspired by the Google File System (GFS). GFS introduced innovative ideas for handling large-scale data, which laid the groundwork for HDFS's design and functionality.

Common Attributes Between GFS and HDFS:

Both GFS and HDFS share several key characteristics, such as their distributed nature and their ability to handle large-scale data efficiently.

Pattern Emphasis: Write Once, Read Many:

Both systems are designed with the "Write Once, Read Many" (WORM) pattern in mind. This approach is well-suited for large datasets that are written once and then read many times, reducing the need for complex data management.

Historical Reference - WORM Drives and Optical Jukebox Libraries:

The WORM pattern is reminiscent of older technologies like WORM (Write Once, Read Many) drives and optical jukebox libraries, which were used before the advent of CDs (CD-R). These technologies allowed data to be written once and read multiple times without alteration.

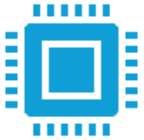
HDFS

Evolution of Storage Technology:

During the era when these technologies were prevalent, WORM optical storage offered a better price-to-performance ratio for larger storage volumes compared to disk technology of that time. Even though disk technology was more common, WORM was still a cost-effective solution, especially when compared to tape storage, which was another popular medium.

Handling Very Large Files:

One of the significant capabilities of HDFS is its ability to manage very large files, with a single file potentially spanning the entire distributed system. This allows for the efficient storage and processing of massive datasets.



Use of Commodity Hardware:

HDFS is designed to run on commodity hardware, which typically includes:

- **Two-socket, 64-bit servers:** Basic server configurations that are affordable and widely available.
- **Local disks:** Instead of relying on specialized storage systems like RAID, HDFS uses local disks on each server.
- **No RAID:** HDFS avoids the complexity and cost of RAID configurations by relying on data replication across multiple nodes for fault tolerance.

Co-location of Data and Compute Resources:

HDFS optimizes performance by co-locating data with compute resources. This means that the data is stored on the same machines that perform computations, reducing the need to transfer large amounts of data over the network and improving overall processing speed.

What It's *Not* Good For

Handling Many Small Files:	The Name Node may face scalability challenges when dealing with a large number of small files. More details to follow.
Low Latency Access:	The focus here is on maximizing throughput (N=XR) rather than minimizing the time it takes to start reading the data (latency).
Multiple Writers & File Updates:	The system only allows one writer at a time. Operations are limited to creating, appending, renaming, moving, and deleting files. There are no in-place updates.
Not a Replacement for Relational Databases:	Data is stored in files without indexing. To locate specific information, all data must be read, usually through a MapReduce job or task.
Unsuitable for Traditional Storage Solutions:	Traditional SAN or NAS systems, like those from NetApp and EMC, are not applicable here.
Outdated Technologies:	This is analogous to programming in COBOL or selling mainframes and FICON technologies, which are considered outdated.

HDFS Concepts & Architecture

Core architectural goal: Fault-tolerance in the face of massive parallelism (many devices, high probability of HW failure)

Monitoring, detection, fast automated recovery

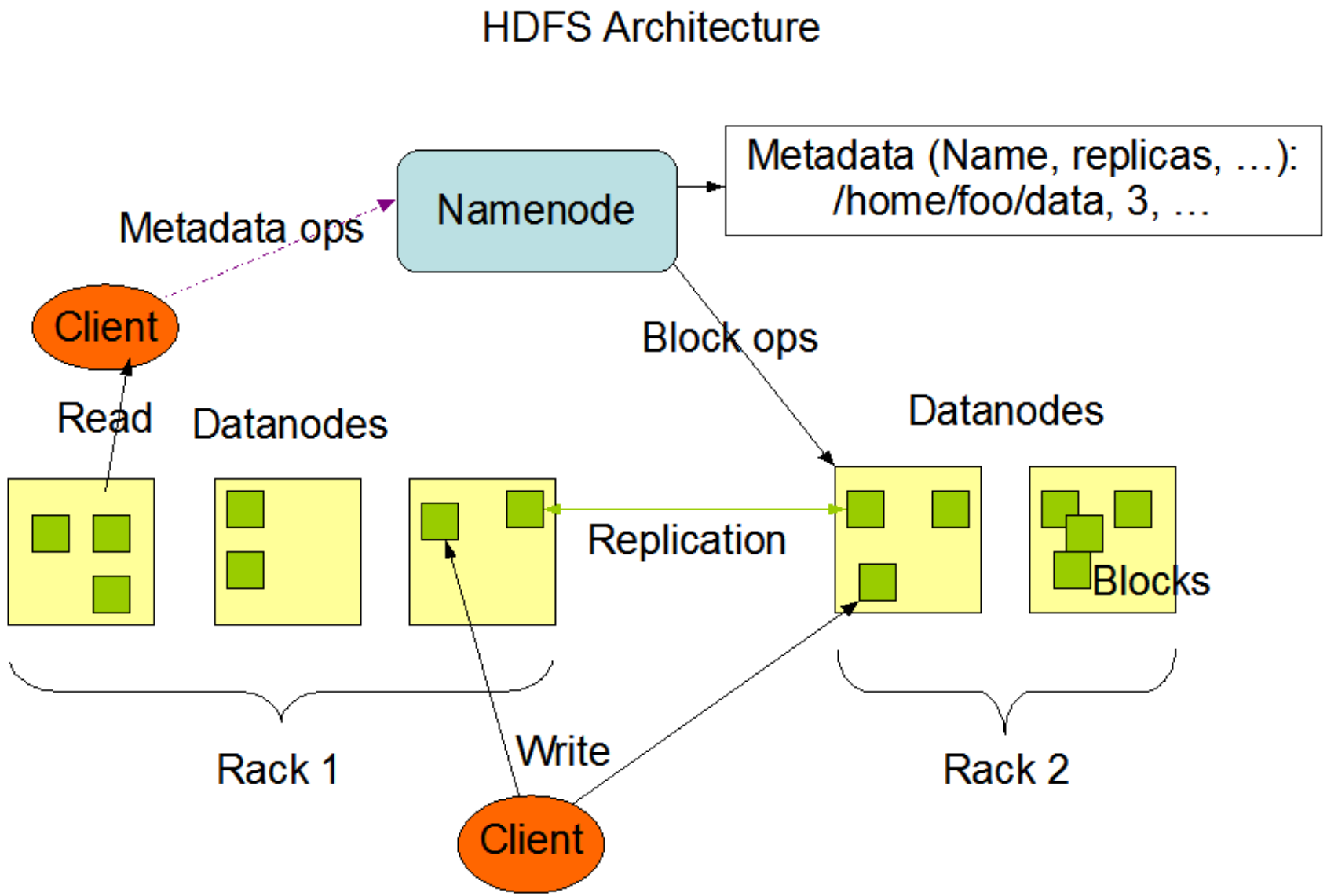
Focus is batch throughput

- Hence support for very large files; large block size; single writer, create or append (no update) and a high read ratio

Master/Slave: Separate filesystem metadata & app data

- One NameNode manages the filesystem and client access
  - All metadata - namespace tree & map of file blocks to DataNodes - stored in memory and persisted on disk
  - Namespace is a traditional hierarchical file organization (directory tree)
- DataNodes manage locally attached storage (JBOD or RAID)

Not layered on or dependent on any other filesystem



http://hadoop.apache.org/common/docs/current/hdfs_design.html

HDFS Architecture

Namenode:

- The **Namenode** is the master server that manages the metadata of the file system. It keeps track of:
  - The filenames
  - The directory structure
  - The locations of the file blocks across the Datanodes
  - The replication factor (how many copies of each block are stored)
- The Namenode does not store the actual data but only the metadata. This includes information such as file paths, permissions, and the mapping of files to blocks.

Datanodes:

- **Datanodes** are the worker nodes that store the actual data blocks of the files.
- Data is split into blocks (typically 128 MB or 256 MB in size) and distributed across multiple Datanodes.
- The Datanodes are responsible for serving read and write requests from clients, and they also handle block creation, deletion, and replication as instructed by the Namenode.
- Blocks are replicated across multiple Datanodes to ensure fault tolerance and high availability.

Client:

- The **Client** interacts with the HDFS by submitting requests to read or write data.
- When a client wants to read a file, it contacts the Namenode to get the metadata (e.g., the locations of the blocks) and then reads the data directly from the Datanodes.
- When writing data, the client first asks the Namenode where to write the blocks, and then writes the data directly to the Datanodes.

HDFS Architecture

Replication:

- HDFS is designed for high fault tolerance. To achieve this, each block of data is replicated across multiple Datanodes (typically three copies, but this can be configured).
- If a Datanode fails, the system will replicate the blocks on the remaining nodes to maintain the required replication factor.
- The diagram shows that data is written by the client to a Datanode, and then replicated to other Datanodes across different racks to ensure availability even if a rack fails.

Racks:

- In the diagram, the Datanodes are organized into **Racks** (Rack 1 and Rack 2). A rack is a collection of physically co-located Datanodes.
- HDFS is aware of the rack topology and places replicas in such a way that at least one replica is on a different rack, providing greater fault tolerance.

Metadata Operations (Metadata ops):

- When a client interacts with HDFS, it first communicates with the Namenode to get the necessary metadata for the operation (e.g. , file location, block locations).
- These operations include checking the location of file blocks and managing the replication of data.

Block Operations (Block ops):

- These are the actual operations performed on the data blocks stored in the Datanodes, such as reading or writing blocks.

HDFS Concepts & Architecture

Block-Oriented Structure:	HDFS (Hadoop Distributed File System) uses a block-oriented storage system where the default block size is 64 MB. The block size can be adjusted on a per-file basis.
Impact of Smaller Block Sizes:	If the block size were significantly smaller, say 512 bytes or 4 KB instead of 64 MB, the metadata required to track these blocks would increase dramatically—by 128,000 or 16,000 times, respectively. This would put immense pressure on the Namenode's memory, possibly limiting the total HDFS storage capacity.
File Size Sensitivity:	The total storage capacity of HDFS is not just affected by block size but also by the total number of files stored in the system.
Block Size Consistency:	All blocks of a file are of the same size, except for the last block, which might be smaller.
Performance Optimization:	The large block size in HDFS reduces the time spent seeking data on the disk. This approach allows the system to achieve disk transfer speeds of up to 100 MB/sec, with disk read-ahead further reducing seek times. The aggregate file read bandwidth can be high, depending on the number of disks per node.
Fault Tolerance through Replication:	HDFS divides files into blocks and stores them across multiple DataNodes with redundancy. The replication factor, which determines the number of copies of each block, is set on a per-file basis. The location of block replicas can change over time to ensure data reliability.

NameNode

Single Point of Failure (SPOF):	The **NameNode** is crucial in the Hadoop Distributed File System (HDFS) as it manages the metadata for the file system. However, it is a single point of failure, meaning that if it fails, the entire HDFS could become inaccessible.
Metadata Storage:	The **FsImage** file stores the entire file system's metadata, which includes the directory structure, file permissions, and block locations.
	Changes to the file system that occur after the last checkpoint (a snapshot of the current state) are recorded in a separate transaction log called the **EditLog**.
	Both the **FsImage** and **EditLog** are stored on the NameNode's local filesystem.
Availability and Redundancy:	To ensure high availability, redundant copies of the **FsImage** and **EditLog** are typically stored on Network-Attached Storage (NAS) or other servers. This way, the data can be recovered even if the NameNode's local storage fails.

NameNode

Checkpointing Process:

Checkpointing refers to the process of merging the **EditLog** with the **FsImage** to create an updated version of the file system's metadata.

This process only occurs when the NameNode is restarted. During startup, the NameNode reads the **EditLog** to update the **FsImage** with recent changes. The updated **FsImage** is then loaded into memory, and the **EditLog** is truncated (i.e., cleared) since its changes have been applied.

Once this process is complete, the NameNode is ready to manage the file system again.

Communication with DataNodes:

The NameNode does not directly send instructions to DataNodes. Instead, it embeds instructions in its responses to the heartbeat signals sent by DataNodes. This includes:

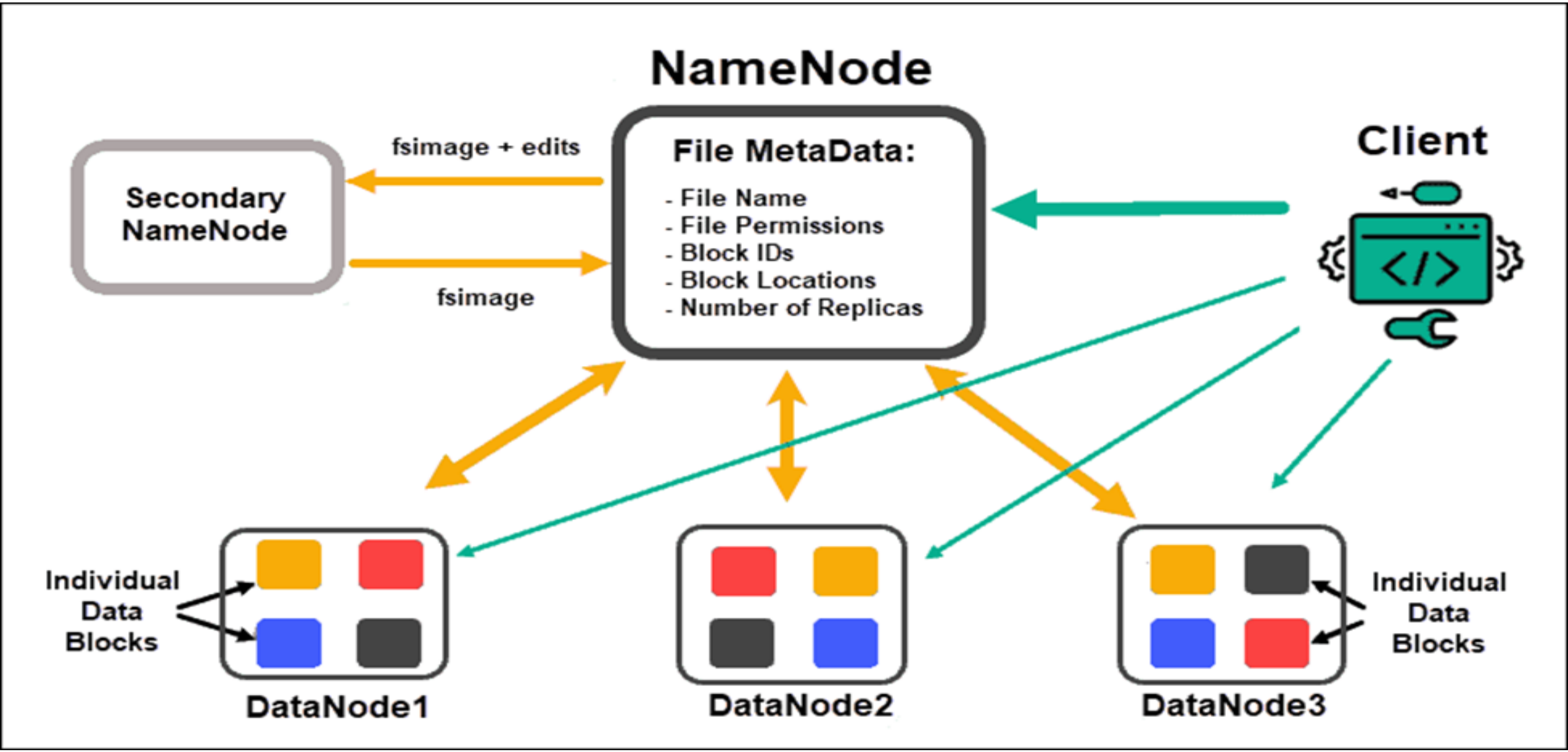
Replication: Directing DataNodes to replicate blocks to other nodes to ensure fault tolerance.

Block Removal: Instructing DataNodes to remove excess block replicas for load balancing.

Node Management: Telling DataNodes to restart (re-register with the NameNode) or to shut down.

Block Reports: Requesting an immediate block report from DataNodes to ensure accurate tracking of block locations.

NameNode


NameNode
File MetaData:
- File Name
- File Permissions
- Block IDs
- Block Locations
- Number of Replicas
Secondary NameNode
fsimage + edits
fsimage
Client
Individual Data Blocks
DataNode1
DataNode2
DataNode3
Individual Data Blocks

DataNodes

In Hadoop, a DataNode is responsible for storing the actual data of the file system. Each block replica on a DataNode consists of two essential files stored in the local file system:

Data File: This file contains the actual data for the block.

Block Metadata: This file holds important information about the block, including its properties and state.

Additionally, each block replica is associated with a block checksum (which ensures data integrity) and a generation timestamp (indicating when the block was created).

Connection with the NameNode

When a DataNode starts up, it establishes a connection with the **NameNode** (which manages the file system namespace and metadata).

During this connection, a **handshake** occurs to verify the DataNode's ID and software version. If either of these does not match, the DataNode shuts down to prevent potential data corruption.

Importantly, DataNodes cannot join the cluster or register with the NameNode unless they possess a valid matching ID.

This mechanism helps maintain the integrity and security of the cluster.

DataNodes

Block Storage Efficiency

- Hadoop optimizes the use of storage space by only consuming the amount needed for the data blocks. It particularly focuses on ensuring that full blocks are utilized whenever possible, except for the last block of a file, which may not be fully filled.

Reporting and Heartbeats

- DataNodes play a crucial role in maintaining communication with the NameNode:

Block Reports:

- Upon startup and subsequently every hour, DataNodes send a **block report** to the NameNode. This report details all the block replicas that the DataNode currently holds.

Heartbeats:

- DataNodes also send a **heartbeat signal** to the NameNode every 3 seconds (by default).
- If a DataNode fails to send a heartbeat within a 10-minute window, it is considered unavailable.
- In such cases, the NameNode removes the DataNode from its records and schedules the creation of new replicas of the blocks that were stored on the failed DataNode on other DataNodes.

Back to the NameNode

Metadata Storage

- The metadata for the file system is stored in two primary files on the NameNode's local file system:
  - **FsImage**: This file contains the complete and persistent snapshot of the filesystem metadata at a given point in time.
  - **EditLog**: This transaction log records all changes made to the filesystem since the last checkpoint. It captures every modification to the metadata, ensuring that no changes are lost.
- Both the FsImage and EditLog files are stored locally on the NameNode, but to enhance availability and reliability, redundant copies can be made and stored on Network-Attached Storage (NAS) or other servers.

Checkpointing Process:

- ***Occurs only at the startup of the NameNode***
- **Startup**: When the NameNode starts, it reads the last saved FsImage from its local filesystem.
- **Replay EditLog**: It then replays the EditLog to apply any changes made since the last checkpoint. This process updates the FsImage to reflect the current state of the filesystem.
- **Memory Mapping**: The updated FsImage is mapped into memory for efficient access and operations.
- **Truncation**: After the EditLog has been successfully replayed and the FsImage is updated, the EditLog is truncated to free up space and improve performance.
- This approach ensures that the NameNode is ready to manage the file system efficiently and effectively as soon as it starts up.

Client Operations in Hadoop

Reading Data

File Location:

- When a client initiates a read request, it first contacts the **NameNode** to obtain the locations of the file's data blocks.
- The NameNode provides the client with a list of DataNodes that hold the replicas of these blocks.

Data Retrieval:

- The client then reads the data blocks from the closest available DataNode(s).
- This proximity helps improve the speed and efficiency of the data transfer.

Data Integrity Check:

- To ensure the data's integrity, the client uses a **block checksum** to detect any corruption that may have occurred during storage or transmission.
- This is particularly important in clusters that contain hundreds to thousands of nodes and disks, where the risk of data corruption is higher.

Client Operations in Hadoop

Writing Data

Write Request:

- For a client write operation, the client sends the file path to the NameNode and requests it to select a set of DataNodes based on the specified **Replication Factor** (the number of copies of each block that need to be stored).

Write Pipeline:

- The data is written to the DataNodes in a **serial pipeline manner**, which ensures efficient data transfer:
  - **Data Bytes**: The client pushes data bytes into the pipeline as a sequence of packets.
  - **Packet Buffering**: The client buffers these bytes until a packet reaches the default size of 64KB or until the file is closed. Once the buffer is filled, the packet is pushed into the write pipeline.
  - **Checksum Calculation**: Along with the data, the client calculates and sends the corresponding checksum for the block to verify its integrity.

Data and Metadata Storage:

- It's important to note that data and metadata (including checksums) are stored separately on the DataNodes, which helps organize and manage the information efficiently.

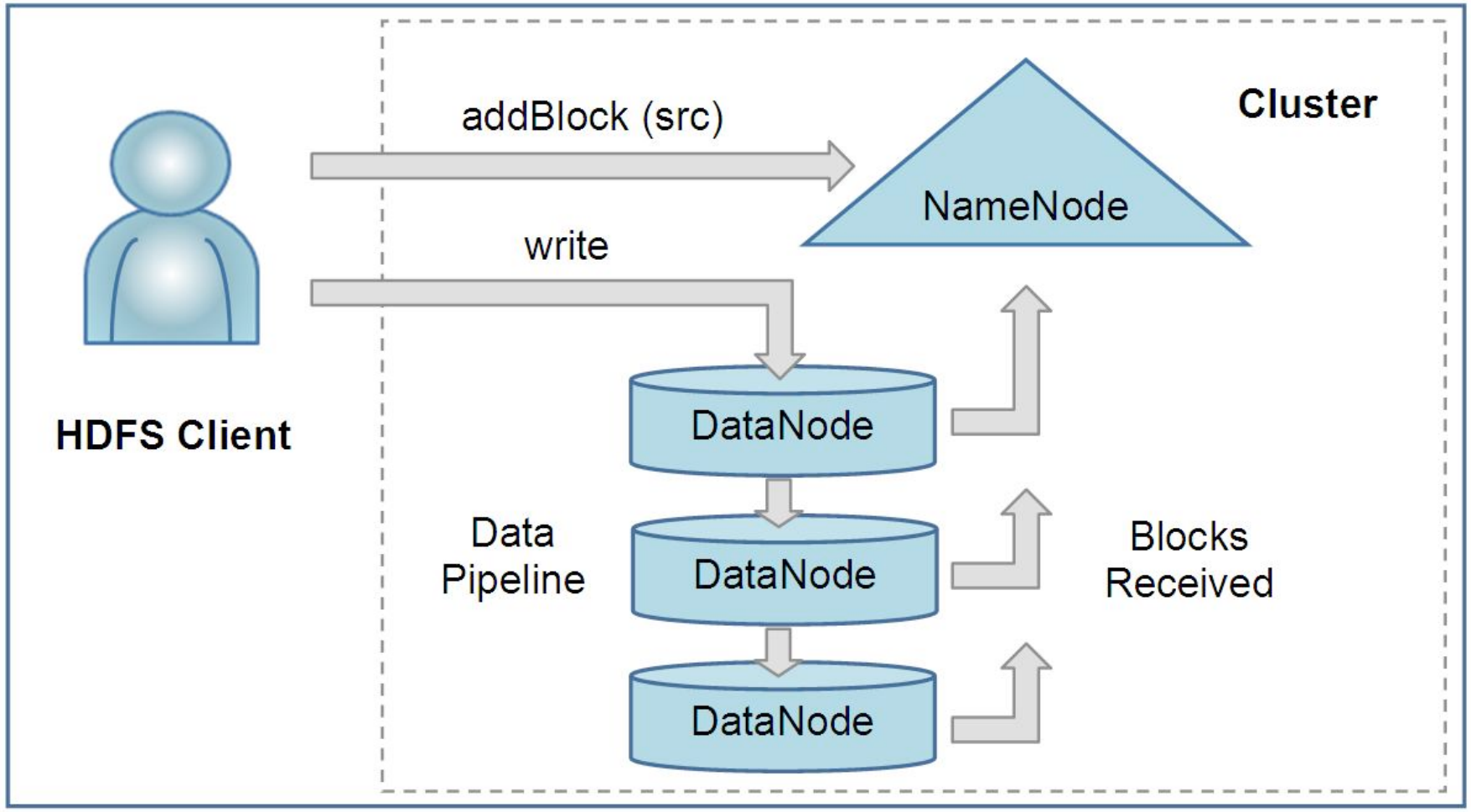
Client Operations in Hadoop

Asynchronous I/O

- The writing process employs **asynchronous I/O**, allowing the client to continue sending packets without waiting for an acknowledgment (ACK) from the DataNodes for each packet.
- The client can send multiple outstanding packets at once, determined by the **packet window size** (the maximum number of packets that can be in transit). This queue size is typically defined as N=XR, where X is the packet size and RRR is the number of outstanding packets.

Data Visibility

- Data becomes visible to new readers only after the file is closed or when the client invokes the **hflush** call, which flushes the buffered data to the DataNodes, making it accessible for reading.



Client Operations in Hadoo

Locality and High Performance in HDFS

The Hadoop Distributed File System (HDFS) differs significantly from traditional non-distributed file systems because its Application Programming Interface (API) reveals the location of a file's data blocks.

This design choice has important implications for performance and data processing.

Data Locality and Computation

Task Scheduling:

- Frameworks like **MapReduce** leverage this feature to schedule tasks that execute close to where the data is stored. This approach allows for:
- **Computation at the Data**:
- Instead of moving data to the computation, the system sends the computation to the data. This minimizes data transfer across the network, which is often a bottleneck in data processing.

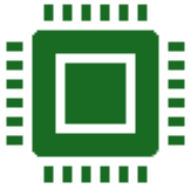
Network Efficiency:

- By executing tasks where the data resides, HDFS reduces the reliance on network infrastructure, leading to:
- **Lower Network Costs**: Less data movement reduces the overall network load.
- **Shorter Processing Times**: Tasks complete faster since they operate directly on local data.

High Performance in HDFS

Read Performance

The highest read performance is achieved when tasks are executed with data located on **local disks**. This local access allows for faster data retrieval and processing.



Storage Technology

HDFS employs **point-to-point SAS (Serial Attached SCSI) or SATA (Serial ATA)** disks, which do not share a bus, making them key enablers of high performance. This design results in:

- **Superior Price-to-Performance Ratio**: HDFS provides a significant price advantage compared to traditional **SAN (Storage Area Network)** storage systems, offering better performance at a lower cost.

Replication and Fault Tolerance

HDFS supports an application-specific replication factor for each file, with the default set to **3**. This feature:

- **Enhances Fault Tolerance**: By keeping multiple copies of each block, HDFS can withstand hardware failures and maintain data availability.
- **Increases Read Bandwidth**: Having multiple replicas improves read performance, especially for files that are heavily accessed.

Typical Hadoop Node

A Hadoop node is a fundamental component in a Hadoop cluster, responsible for processing and storing data.

The configuration of a typical Hadoop node is designed to meet the demands of big data applications and is optimized for performance, storage capacity, and cost-effectiveness.

A typical Hadoop node is carefully configured to handle the requirements of big data processing and storage.

With its dual socket architecture, quad-core CPUs, significant memory, and multiple data disks, it is optimized for performance and efficiency in a distributed environment.

This setup not only supports heavy querying but also allows for scalable data storage solutions, making it suitable for a wide range of applications from real-time analytics to long-term data archiving.

Typical Hadoop Node - Hardware C

Dual Socket 1U/2U Rack Mount Server:	The server is designed to fit into standard data centre racks, either in a inches tall) form factor. This compact design maximizes space utilizatic physical space is a constraint.
Internal Disk Requirement:	Due to the need for multiple internal disks to store large amounts of data, blade servers are not suitable for typical Hadoop nodes. Blade servers are designed for high density and may lack the necessary storage capacity to support Hadoop's distributed storage model effectively.
Quad-Core CPUs:	Each node is equipped with quad-core CPUs operating at speeds greater than 2 GHz. This configuration allows the node to handle multiple threads simultaneously, improving the performance of data processing tasks, such as running MapReduce jobs.
Memory (16-24 GB):	With 16 to 24 GB of RAM, the node can efficiently manage data processing tasks and operations. This amount of memory helps in caching data, reducing read times from disk, and improving overall system performance during high-demand scenarios.
Data Disks (4 to 6 @ 1TB):	The node typically contains 4 to 6 data disks, each with a capacity of 1TB. This configuration allows for substantial local data storage, facilitating high-speed access to data during processing. The distributed nature of Hadoop enables data to be split into blocks, with these blocks stored across the various disks.

Typical Hadoop Node - Performance and Economic Efficiency

Performance Target:

- The node's hardware is specifically targeted towards workloads involving heavily queried data.
- It is designed to efficiently process large volumes of data and execute complex queries in a timely manner, benefiting from the combination of fast CPUs and ample disk space.

Economic Efficiency:

- The economic aspect of this node configuration is crucial, as it balances performance with cost.
- By utilizing commercially available components in a standard server format, organizations can achieve significant price-to-performance benefits compared to more specialized hardware solutions.

Typical Hadoop Node - Use Cases, Networking & OS

Long-Term Storage Considerations:

For use cases that prioritize long-term data storage, organizations might opt for a **DL370 5U form factor** server.

This type of server supports 14 internal 3.5” disks, providing extensive storage capacity for archiving large datasets over extended periods.

This configuration is suitable for organizations that need to store historical data while also ensuring easy access for future analysis.

Networking

Each node is equipped with a **1 GbE NIC (Gigabit Ethernet Network Interface Card)**, enabling data transfer rates of up to 1 Gbps.

This speed is adequate for most Hadoop workloads, allowing for efficient data communication between nodes in the cluster.

However, as data demands grow, upgrading to faster networking options, such as 10 GbE, may be necessary to avoid bottlenecks.

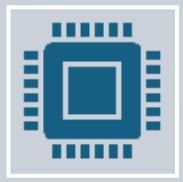
Operating System:

The typical operating system for a Hadoop node is **Linux**, which is preferred for its stability, security, and flexibility.

Linux provides a robust platform for running Hadoop, with extensive support for various distributions that cater to different needs in terms of performance tuning and management.

Hadoop MapReduce

Imagine a massive dataset, such as all Google search queries for a day.



Processing such a vast amount of data using traditional methods is inefficient and time-consuming.

Hadoop MapReduce offers a solution by breaking down the problem into smaller, manageable tasks that can be processed concurrently across multiple computers.

How Hadoop MapReduce Works

Data Division:

The massive dataset is divided into smaller chunks called blocks.

These blocks are distributed across multiple computers in a cluster.

Map Phase:

Each computer independently processes its assigned blocks.

A user-defined map function is applied to each record in the block.

The map function transforms the input data into intermediate key-value pairs.

Example: If processing search queries, the map function might extract the search term as the key and a count of 1 as the value.

Shuffle and Sort:

Intermediate key-value pairs from all map tasks are collected.

The framework groups together values with the same key.

This phase ensures that all values associated with a particular key are together.

Reduce Phase:

The grouped data is processed by reducer tasks.

A user-defined reduce function combines the values for each key.

The reduce function produces the final output.

Example: For search queries, the reduce function would sum up the counts for each search term to determine the total frequency.

Let's consider a simple word count example to illustrate the MapReduce process:

- **Input data:** A text file containing multiple lines of words.
- **Map function:**
  - Input: A key-value pair where the key is the offset in the file and the value is a line of text.
  - Output: A key-value pair where the key is a word and the value is 1.

Reduce function:

- Input: A key (word) and a list of values (counts).
- Output: A key-value pair where the key is the word and the value is the total count of occurrences.

Hadoop Implementation:

- Use the Hadoop API to define the map and reduce functions as Java classes.
- Configure the MapReduce job with input/output formats, number of mappers/reducers, etc.
- Submit the job to the Hadoop cluster for execution.

Real-Time Example: Analyzing Search Queries

Let's say you want to find the most popular search term on Google on a particular day. Here's how Hadoop MapReduce would tackle it:

Divide and Conquer	Map	Shuffle and Sort	Reduce
Divide and Conquer: The massive dataset of search queries is split into smaller chunks, each assigned to a different computer in the cluster.	Map Phase: Each computer processes its chunk and counts the occurrences of each search term. This step is like tallying the votes in an election.	Shuffle and Sort: The results from all computers are combined and sorted by search term. This is similar to organizing the vote tallies from different polling stations.	Reduce Phase: The sorted data is further processed to determine the most frequent search term. This is like counting the final votes to determine the winner.

Advantages of Hadoop MapReduce

Scalability: Handles massive datasets by distributing processing across multiple machines.

Fault Tolerance: Automatically recovers from failures by replicating data and tasks.

Efficiency: Processes data in parallel, improving performance.

Simplicity: Provides a high-level abstraction for complex data processing tasks.

Practical Applications of Hadoop MapReduce

Log Analysis:	Processing vast amounts of server logs to identify trends, anomalies, and potential security threats.
Web Analytics:	Analyzing website traffic patterns, user behavior, and clickstream data to optimize user experience and marketing campaigns.
Recommendation Systems:	Processing user data to suggest products, movies, or content based on preferences and behavior.
Fraud Detection:	Analyzing financial transactions to identify patterns indicative of fraudulent activities.
Scientific Computing:	Processing large datasets in fields like genomics, astronomy, and climate modeling.
Social Media Analysis:	Analyzing social media data to understand sentiment, trends, and user interactions.

Performance Optimization in Hadoop MapReduce

Data Locality:	Ensuring data is processed on the same node where it's stored to minimize data transfer.
Input Format and Output Format:	Choosing the appropriate input and output formats based on data characteristics.
Combiner:	Performing partial aggregation within the map phase to reduce data transferred to reducers.
Partitioner:	Controlling how data is distributed to reducers based on keys.

Performance Optimization in Hadoop MapReduce

Number of Mappers and Reducers:

Adjusting the number of mappers and reducers to balance workload and resource utilization.

Data Compression:

Compressing input and output data to reduce storage and network overhead.

Hardware Optimization:

Leveraging faster processors, more memory, and SSDs to improve performance.

Language Support

MapReduce is a programming model used to process large data sets with a distributed algorithm on a cluster.

Java natively supports the MapReduce framework, making it a good fit for stream processing and handling large-scale text processing tasks.

The framework allows the user to write Map and Reduce functions that process data in a pipeline, where the Map function's output (key-value pairs) is sorted and then passed to the Reduce function.

Python, Perl, and Ruby are also viable for MapReduce but typically use Unix pipes for processing. They are known for their text processing capabilities.

C++ can also be used but requires a different approach, involving socket programming, which deals with network communication rather than standard input/output streams. This approach is less common than the native Java or scripting language implementations.

Master/Slave Architecture in Hadoop's MapReduce:

Master/Slave Architecture:

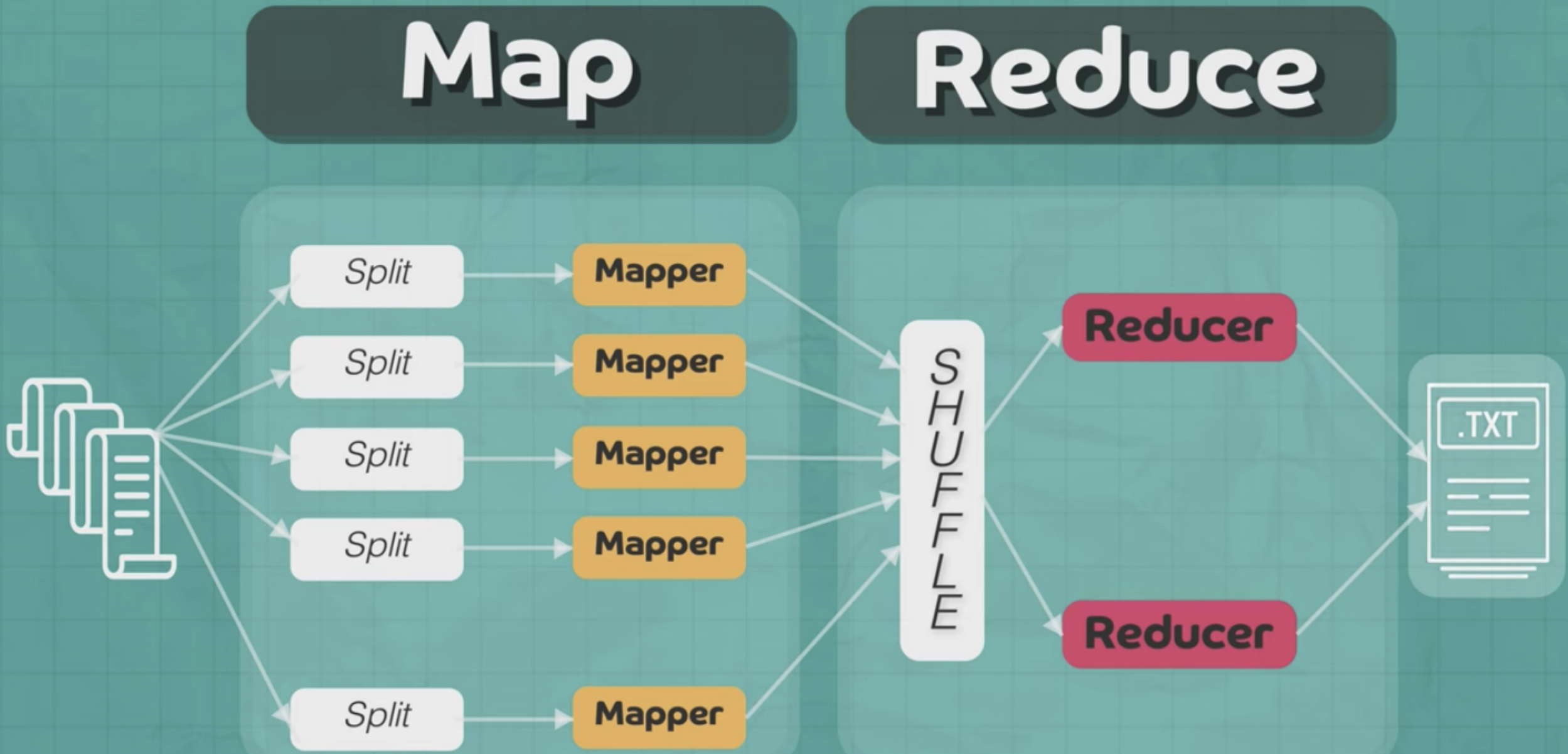
- Hadoop's MapReduce framework uses a **Master/Slave architecture** to manage and execute distributed data processing tasks.

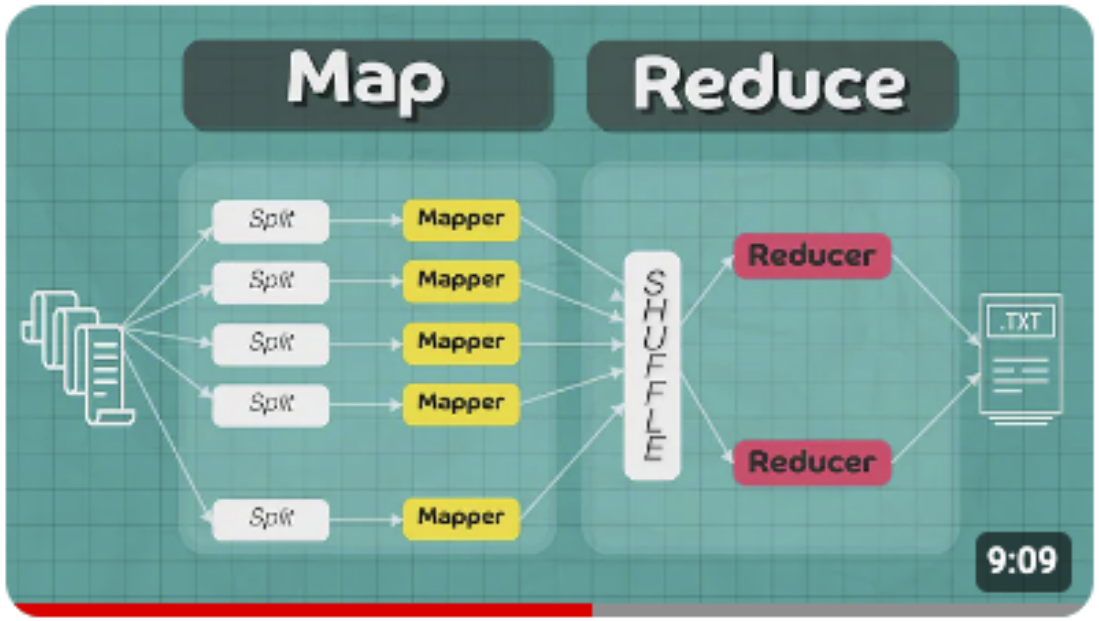
Job Tracker (Master):

- The **JobTracker** acts as the **master** in the cluster.
- There is only **one JobTracker per cluster**.
- The JobTracker's main responsibilities are:
  - **Scheduling**: It schedules the tasks of a job across the cluster. Each job is divided into smaller tasks, which are then assigned to different nodes for execution.
  - **Monitoring**: It continuously monitors the tasks assigned to various nodes. If any task fails, the JobTracker reassigns that task to another node to ensure successful completion.

TaskTracker (Slave):

- Each DataNode in the cluster has a **TaskTracker**, which acts as the **slave**.
- The TaskTracker's role is to:
  - **Execute Tasks**: It performs the actual execution of tasks as directed by the JobTracker.
  - It ensures that the tasks are completed as per the instructions and reports back to the JobTracker with the status of task execution.


Map
Reduce
Split
Split
Split
Split
Split
Mapper
Mapper
Mapper
Mapper
Mapper
SHUFFLE
Reducer
Reducer
.TXT



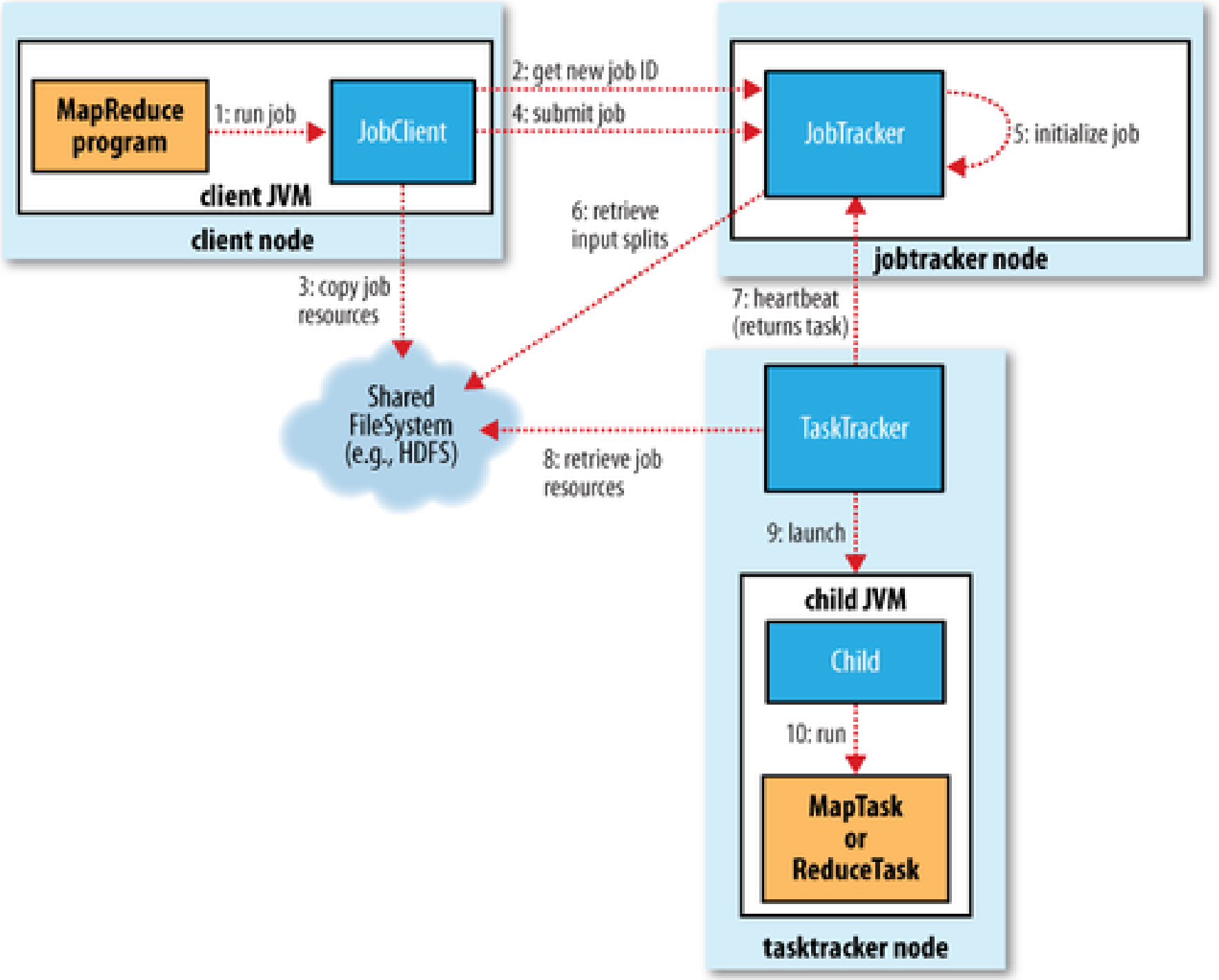
Map Reduce explained with example | System Design

142K views • 1 year ago

ByteMonk

In this video I explain the basics of **Map Reduce** model, an important concept for any software engineer to be aware of. This...

4K


MapReduce program
1: run job
JobClient
client JVM
client node
2: get new job ID
4: submit job
JobTracker
5: initialize job
jobtracker node
6: retrieve input splits
3: copy job resources
7: heartbeat (returns task)
Shared FileSystem (e.g., HDFS)
8: retrieve job resources
TaskTracker
9: launch
child JVM
Child
10: run
MapTask or ReduceTask
tasktracker node

The MapReduce Job Submission Process:

Client Submits Job:

A **MapReduce program** written by the client submits a job to the cluster through the JobClient.runJob API.

The **JobClient** is an instance running in the client's Java Virtual Machine (JVM) on the client node.

Job Request Sent to JobTracker:

The JobClient sends a request to the **JobTracker**, which resides on a dedicated JobTracker node within the cluster.

The JobTracker is responsible for coordinating the execution of the job across the cluster.

Job ID Assignment:

Upon receiving the job request, the JobTracker assigns a unique **Job ID** to the job. This ID is used to track and manage the job throughout its lifecycle.

Data Splitting by JobClient:

The JobClient calculates the **splits** of the input data. These splits represent chunks of the data that will be processed by individual Map tasks.

The JobClient then **fans the job out** to multiple systems by preparing it for distributed execution across the cluster.

Program and Data Distribution:

The JobClient copies the necessary program components, such as the Java JAR file containing the MapReduce code and the configuration file, along with the calculated data splits to the JobTracker's filesystem.

By default, the JobClient writes 10 copies of this data to 10 different JobTrackers for redundancy and reliability. It's important to note that there is usually only one JobTracker per DataNode.

The MapReduce Job Submission Process:

Job Queueing and Initialization:

The JobTracker inserts the job into its internal **job queue**.

This queue is managed by the JobTracker's scheduler, which dequeues jobs and initializes them when resources are available.

During initialization, the JobTracker creates a **job object**.

This object represents the various tasks that need to be executed and keeps track of their progress and status.

Job Execution Notification:

The JobClient informs the JobTracker that the job is now ready to execute.

This triggers the JobTracker to move forward with task assignment.

Task Creation and Data Retrieval:

The JobTracker creates a list of **tasks** that need to be run as part of the job.

It retrieves the input data splits computed by the JobClient from the Hadoop Distributed File System (**HDFS**).

For each data split, the JobTracker creates a corresponding **Map task** that will process that particular piece of data.

Map Task Execution:

The Map tasks are distributed across the cluster and executed in parallel on the available TaskTrackers (slaves).

The output of these Map tasks is then processed by the Reduce tasks, leading to the final output of the MapReduce job.

TaskTrackers Communication with JobTracker:

Periodic Heartbeat Messages:

- TaskTrackers, which are slave nodes in the Hadoop cluster, communicate with the JobTracker via periodic heartbeat messages.
- These messages serve two main purposes:
  - **Indicate that the TaskTracker is Alive**:
    - The heartbeat confirms that the TaskTracker is operational and part of the cluster.
  - **Availability Status**:
    - The heartbeat message also informs the JobTracker whether the TaskTracker is available to run new tasks.

JobTracker Response:

- **Task Assignment**:
- If a TaskTracker is available to run a task, the JobTracker includes a task assignment in its heartbeat response message.
- This ensures efficient utilization of cluster resources by distributing tasks across available TaskTrackers.

TaskTrackers' Task Management:

Limited Queue Size:

- Each TaskTracker has a limited queue size for both Map and Reduce tasks.
- This queue size is determined by the number of CPU cores and the amount of physical memory available on the TaskTracker node.

Task Scheduling:

- The TaskTracker’s internal scheduler prioritizes the execution of Map tasks over Reduce tasks.
- This approach helps ensure that data is processed as early as possible in the pipeline.

Task Preparation and Execution:

Copying Jar File:

- Once a task is assigned, the TaskTracker retrieves the necessary jar file containing the MapReduce code from the Hadoop Distributed File System (HDFS).
- This jar file is copied to the TaskTracker's local filesystem.

Unpacking the Jar File:

- The jar file is then unpacked into a local working directory specifically created for the task.
- This directory contains all the resources needed to execute the task.

TaskRunner Creation:

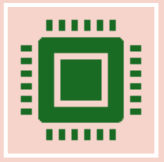
- A TaskRunner instance is created to manage the execution of the task.
- This component is responsible for the actual running of the Map or Reduce task on the node.

Starting a JVM for Each Task:

- The TaskRunner starts a new Java Virtual Machine (JVM) for each task.
- By running each task in a separate JVM, the TaskTracker ensures that the main TaskTracker process is isolated from any potential bugs or errors in the user's MapReduce program.

Isolation and Reliability

Separate JVMs: Running each task in its own JVM enhances the reliability and stability of the Hadoop cluster.



If a user’s MapReduce program contains bugs or causes a crash, it will only affect that specific JVM instance and not the entire TaskTracker node or other running tasks.

Task Progress Reporting in Hadoop MapReduce

TaskTracker Per-Task JVM (Children) Communication:

- **Umbilical Interface**: Each task within a MapReduce job runs in a separate Java Virtual Machine (JVM), referred to as a "child" process.
- These child JVMs communicate with the parent TaskTracker via an “umbilical” interface. This interface is responsible for sending regular updates on the status and progress of the task every few seconds.

Streaming Interface:

- For tasks using the Hadoop Streaming API, the streaming interface handles communication between the TaskTracker and the task process using standard input (stdin) and output (stdout) streams.
- This is typically used when MapReduce jobs are written in languages other than Java, such as Python or Ruby.

Importance of Status and Progress Communication:

- **Long-Running Jobs**: MapReduce jobs can run for extended periods, ranging from minutes to several hours. Because of this, it is crucial for the TaskTracker and JobTracker to monitor the status and progress of each task to ensure smooth execution and timely intervention if needed.
- **State Reporting**: Tasks report their state to the TaskTracker, which could be one of the following:
  - **Running**: The task is currently being executed.
  - **Successful Completion**: The task has completed its execution successfully.
  - **Failed**: The task has encountered an issue and failed.
- **Progress Reporting**: The progress of the task is communicated, indicating how much of the work has been completed. This includes the progress of both the Map and Reduce phases of the job.

Task Progress Reporting in Hadoop MapReduce

Job Counters and User-Defined Status Messages:

- **Job Counters**: Hadoop uses counters to measure various metrics during the execution of a job, such as the number of processed records, bytes read/written, etc. These counters are essential for monitoring the job's progress and performance.
- **User-Defined Status Messages**: Developers can define custom status messages that are reported back to the TaskTracker. These messages can provide more granular insights into the specific operations being performed by a task.

Defining Task Progress:

- **Input Record Read**: For Map tasks, progress can be measured by the number of input records read from the data split.
- **Output Record Written**: For both Map and Reduce tasks, the progress can also be measured by the number of output records written.
- **Reporter Interface**: Hadoop provides a Reporter interface that tasks can use to report progress and update status messages. The task can:
  - **Set Status Description**: Update the status description that is visible in the job tracker UI.
  - **Increment Counters**: Use the Reporter to increment custom or built-in counters.
  - **Call progress() Method**: Explicitly call the progress() method on the Reporter to indicate that the task is making progress and prevent the TaskTracker from assuming that the task has hung or failed.

Job Completion and Cleanup in Hadoop MapReduce

JobTracker Receives Last Task's Completion Notification:

- **Task Completion**: As tasks in a MapReduce job finish their execution, they notify the JobTracker of their completion. When the final task in the job finishes and sends this notification, the JobTracker knows that the entire job is done.

Job Status Changed to "Successful":

- **JobTracker's Role**: Upon receiving the last task's completion notification, the JobTracker updates the job's status to "successful." This indicates that all tasks have been executed without errors and that the job has successfully processed the input data.

HTTP Job Notification to JobClient:

- **Notification Mechanism**: The JobTracker can send an HTTP notification to the JobClient to inform it that the job has been successfully completed. This is an optional mechanism that can be configured depending on the setup.

Job Completion and Cleanup in Hadoop MapReduce

JobClient Polling and Success Notification:

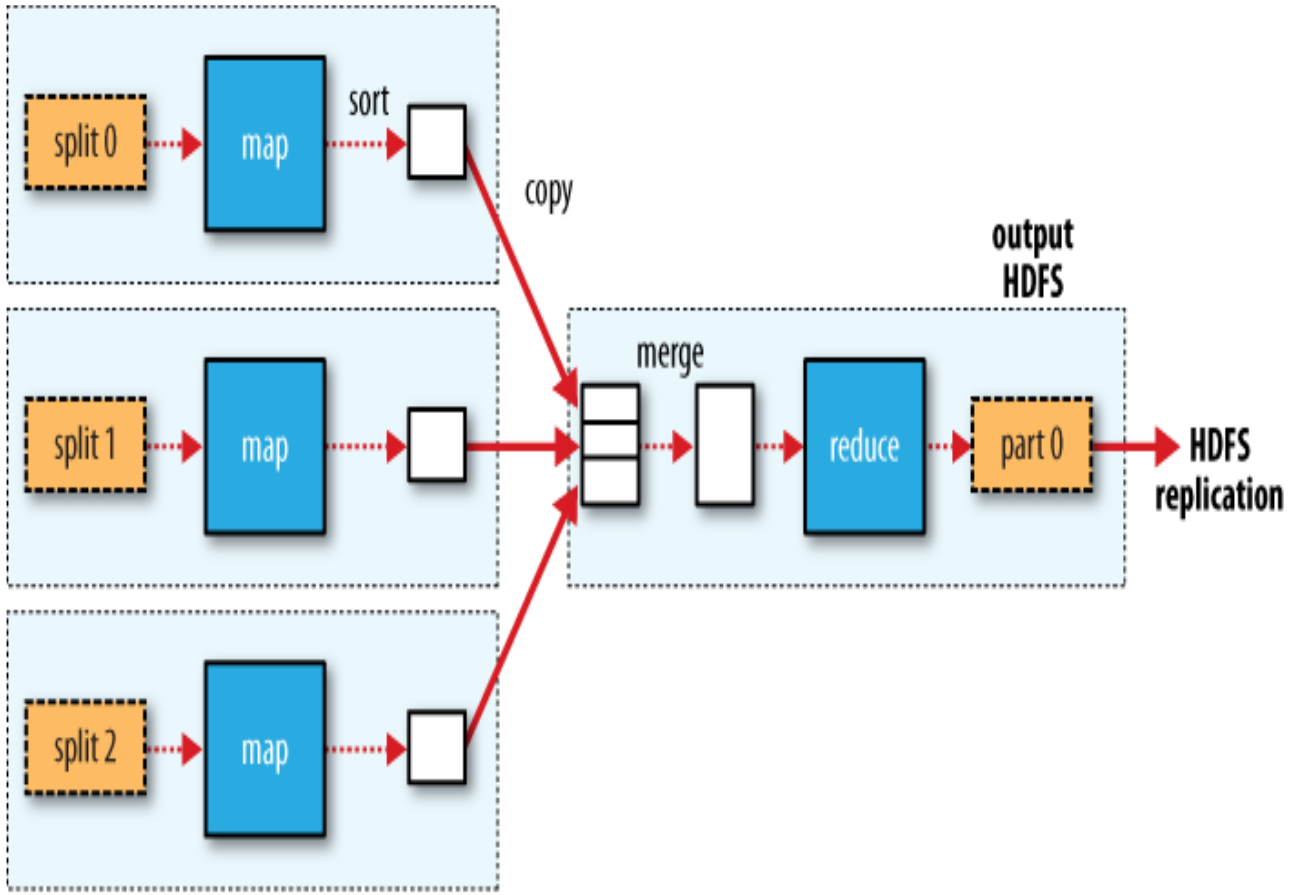
- **Polling for Status**: Even if the JobClient does not receive an HTTP notification, it regularly polls the JobTracker to check on the status of the submitted job. During its next polling interval, the JobClient will learn that the job has completed successfully.
- **User Notification**: After detecting the job's success, the JobClient notifies the user, typically via a console message or an application-specific alert, depending on the environment.

JobTracker Cleanup and TaskTracker Cleanup Commands:

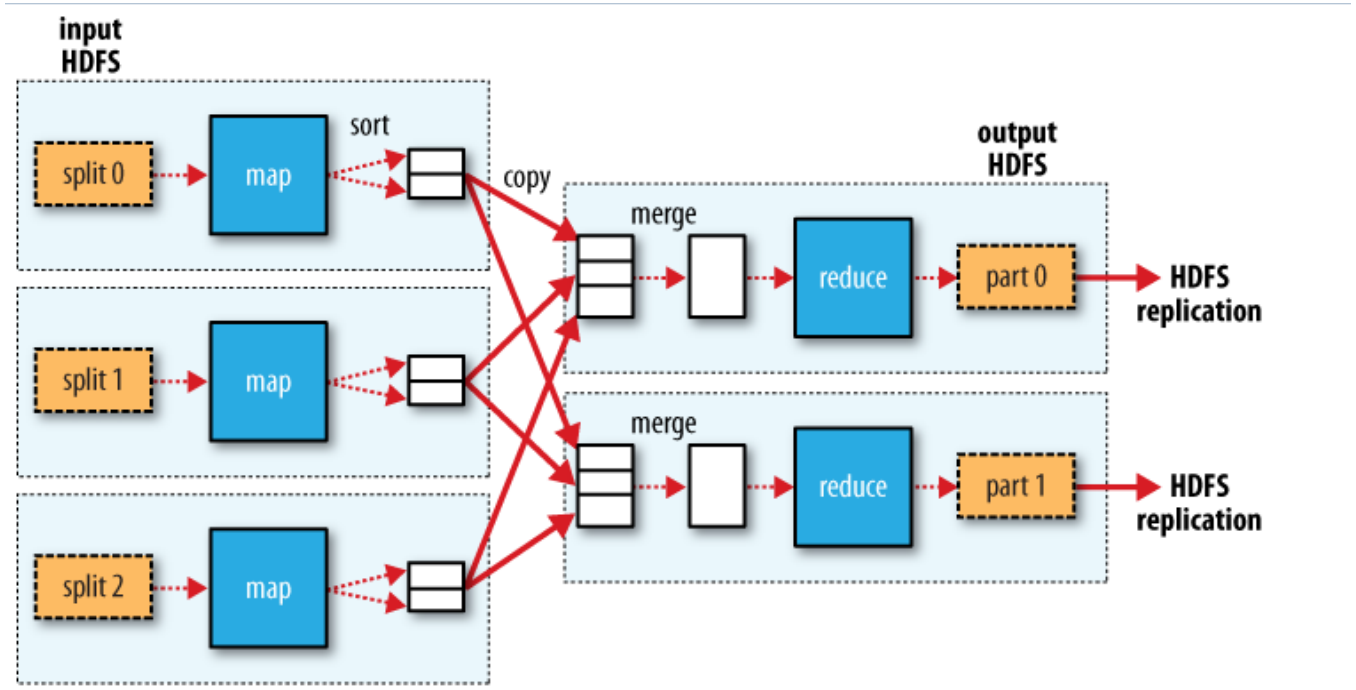
- **Post-Completion Cleanup**: Once the job has been marked as successful, the JobTracker initiates a cleanup process. This involves:
  - **Removing Temporary Files**: The JobTracker commands the TaskTrackers to clean up any temporary files or data that were created during the execution of the job. These might include intermediate Map outputs, temporary working directories, and other files stored on the TaskTracker nodes.
  - **Releasing Resources**: The JobTracker releases the resources allocated to the job, making them available for other jobs in the queue.
- **Commands to TaskTrackers**: The JobTracker sends specific cleanup commands to the TaskTrackers to ensure that all associated resources are freed, and the system is ready for new jobs.

Example Job & Filesystem Interaction

- Multiple Map tasks, one Reduce task
- Map reads from HDFS, writes to local FS
- Reduce reads from local FS, writes to HDFS



Example # Job & Filesystem Interaction
Multiple Map tasks,
Two Reduce tasks
Each Reduce writes one partition per Reduce task to HDFS



Apache Spark

SPARK CORE: THE FOUNDATION OF THE APACHE SPARK FRAMEWORK PROVIDING IN-MEMORY COMPUTING CAPABILITIES.

SPARK SQL: MODULE FOR WORKING WITH STRUCTURED DATA, PROVIDING AN INTERFACE SIMILAR TO SQL.

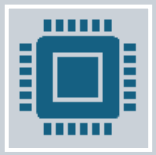
SPARK STREAMING: ENABLES SCALABLE AND FAULT-TOLERANT STREAM PROCESSING OF LIVE DATA STREAMS.

MLLIB: MACHINE LEARNING LIBRARY FOR SPARK OFFERING VARIOUS ALGORITHMS AND UTILITIES.

GRAPHX: API FOR GRAPHS AND GRAPH-PARALLEL COMPUTATION.

NoSQL Databases

MongoDB: A document-oriented NoSQL database used for high-volume data storage.



Cassandra: A highly scalable NoSQL database designed for handling large amounts of data across many commodity servers.

Couchbase: A NoSQL database that provides distributed data management with built-in caching, indexing, and querying.

Data Warehousing Solutions

Amazon Redshift: A fully managed data warehouse service in the cloud.

Google BigQuery: A serverless, highly scalable, and cost-effective multi-cloud data warehouse.

Snowflake: A cloud-based data warehousing service known for its scalability and ease of use.

Data Integration Tools

Apache Nifi: An open-source data integration tool that supports powerful and scalable directed graphs of data routing, transformation, and system mediation logic.

Talend: Provides tools for data integration, data management, enterprise application integration, and big data.

Data Visualization Tools

Tableau: Offers interactive data visualization products focused on business intelligence.

Power BI: Microsoft's suite of business analytics tools for data analysis and sharing insights.

D3.js: A JavaScript library for producing dynamic, interactive data visualizations in web browsers.

Machine Learning Platforms

TensorFlow: An open-source library for dataflow and differentiable programming across a range of tasks.

Scikit-learn: A free software machine learning library for the Python programming language.

PyTorch: An open-source machine learning library based on the Torch library, used for applications such as computer vision and natural language processing.

Cloud-Based Solutions

Amazon Web Services (AWS): Provides a comprehensive suite of cloud computing services including storage, databases, analytics, and machine learning.

Google Cloud Platform (GCP): Offers a range of cloud services for computing, storage, data analytics, and machine learning.

Microsoft Azure: A cloud computing service created by Microsoft for building, testing, deploying, and managing applications and services through Microsoft-managed data centers.