W02 – DATA STORAGE & PROCESSING IN DISTRIBUTED SYSTEMS;

APACHE SPARK;

THE HOANG

LEARNING OBJECTIVES – W02

- Data Processing & Storage in Distributed Systems
- Spark Architecture & Core Components

DATA PROCESSING

- Understand the difference between batch processing and stream processing.
- Learn key concepts and architectures of both processing paradigms.
- Explore real-world examples of batch and stream processing.
- Discuss when to use batch processing vs. stream processing.

Data processing - Introduction to data processing

Definition

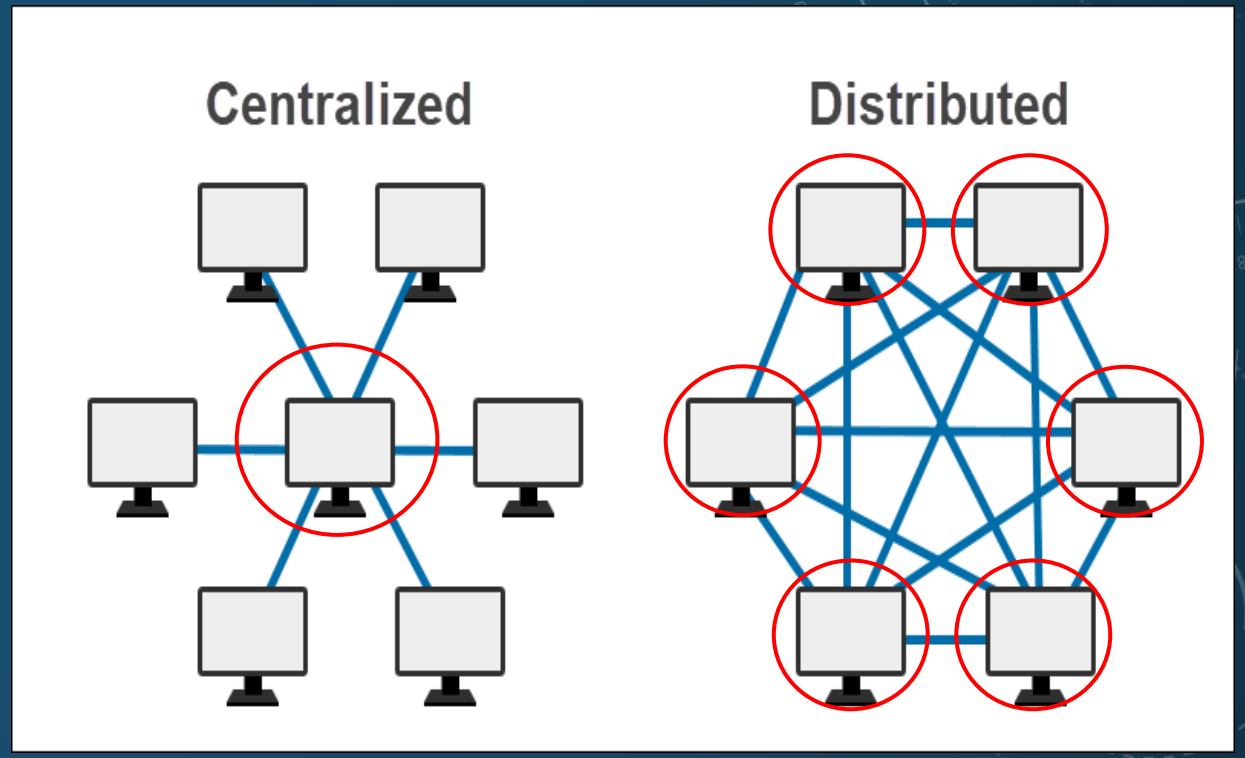
Data processing refers to the collection, manipulation, and processing of data to generate meaningful information.

Centralized vs. Distributed processing:

- Centralized: data processing occurs in a single location.
- Distributed: data processing is spread across multiple machines or nodes.

Importance in distributed systems:

- Scalability: handle large volumes of data.
- Fault tolerance: resilience to failures.



Data processing - What is Batch Processing?

Definition

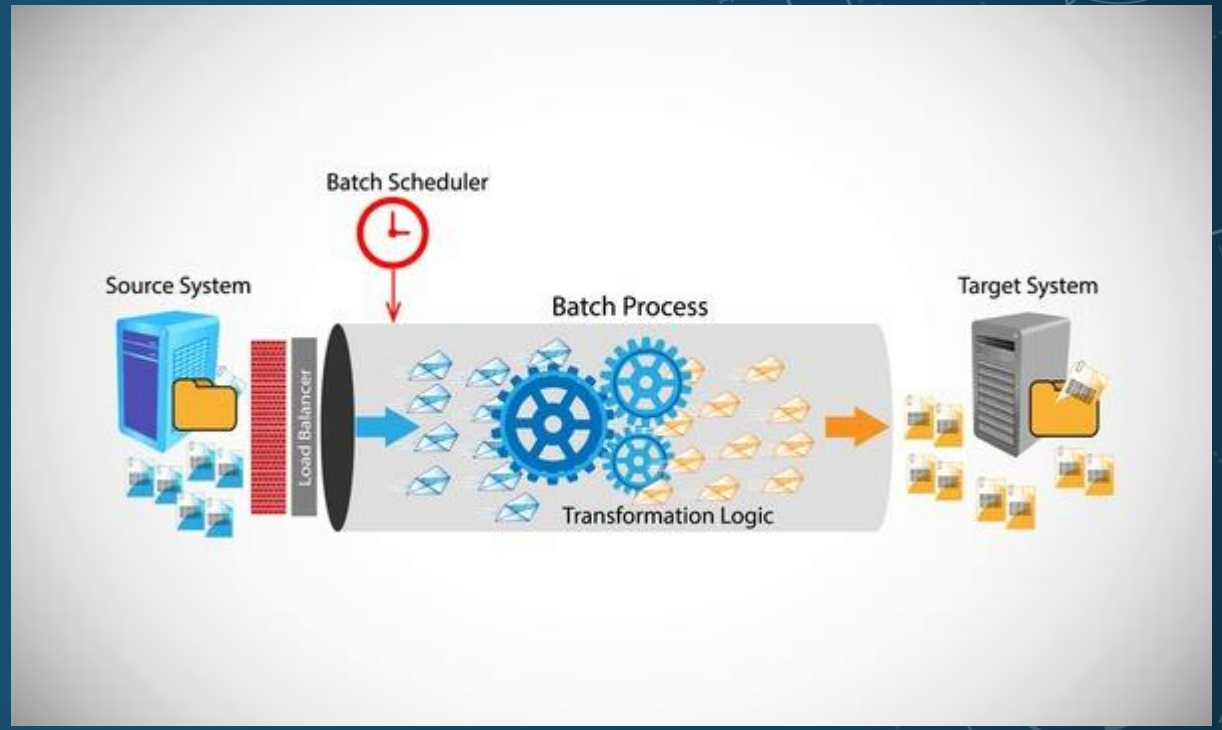
Batch processing involves processing large volumes of data in chunks or batches at scheduled intervals.

Characteristics

- Data is collected over time and processed as a unit.
- Suitable for processing historical data.
- High throughput but higher latency.

Example

Using Hadoop MapReduce to analyze log files collected over a month.



Data processing – Batch processing example

Scenario

Analyzing web server log files collected over the last month to generate a report on user activity.

Steps

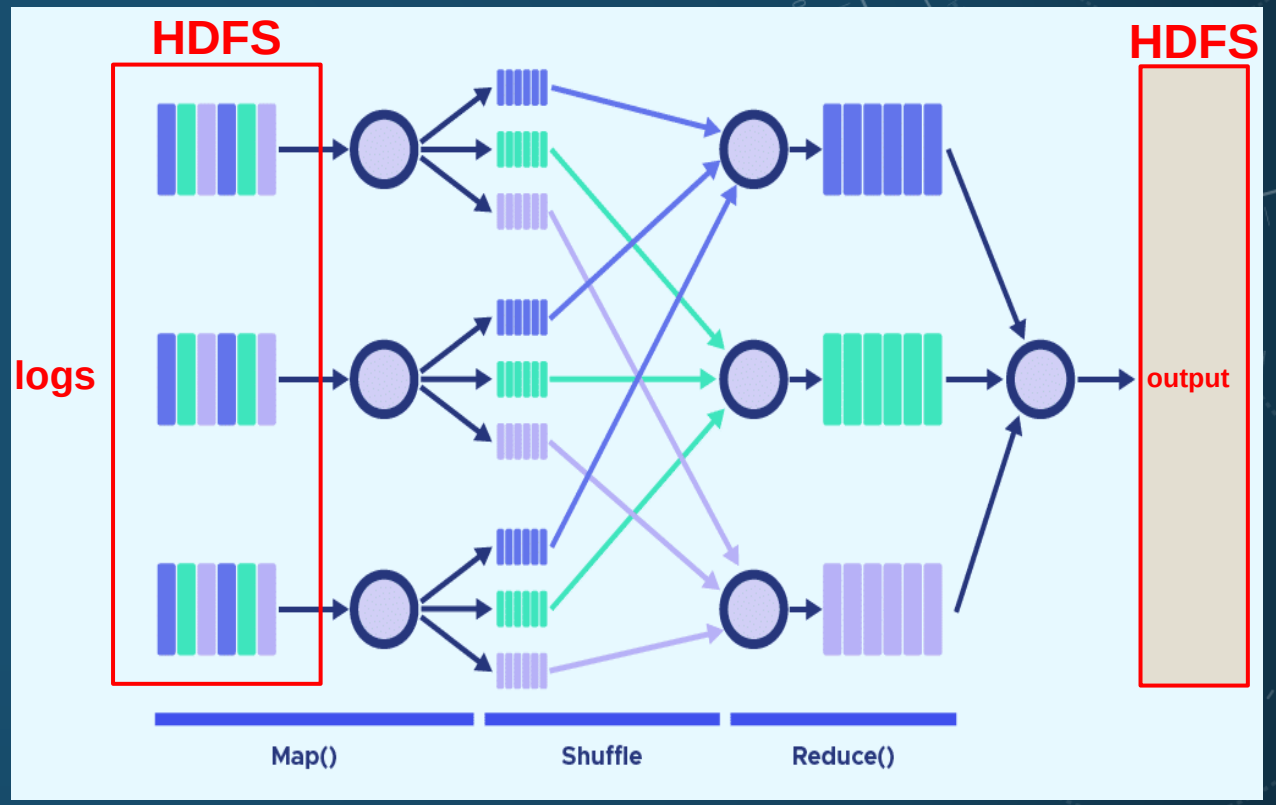
- Data Collection: Logs are stored in HDFS.
- Map Phase: Each log file is processed to extract relevant information (e.g., user ID, timestamp).
- Reduce Phase: Aggregated results are generated (e.g., total visits per user).
- Output: A report on user activity for the month.

Advantages

High throughput for large datasets.

Disadvantages

Not suitable for real-time data needs.



Data processing – What is Stream Processing?

The term "stream" in data processing refers to a continuous flow of data. This is similar to how a physical stream or river constantly flows with water.

Data processing – What is Stream Processing?

Definition

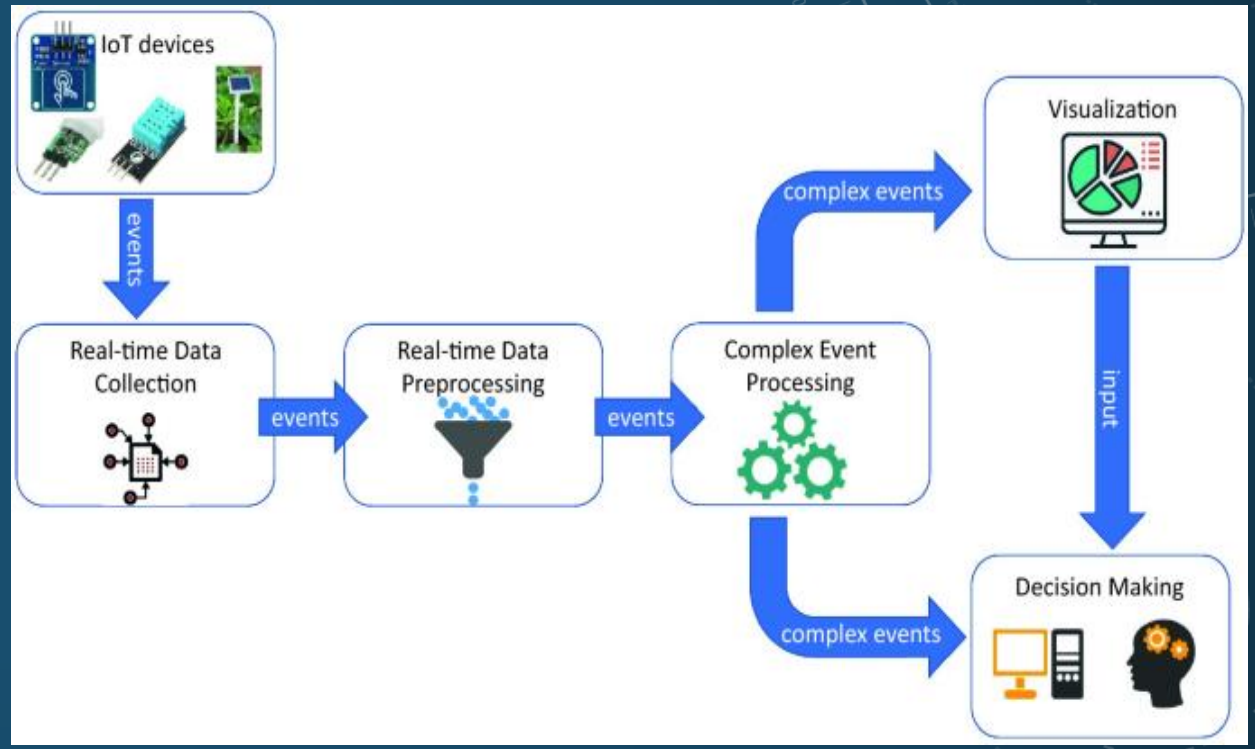
Stream processing involves processing data in real-time as it arrives, enabling low-latency processing.

Characteristics

- Continuous input and processing of data.
- Unbounded, they have no predefined end.
- Suitable for real-time analytics and monitoring.
- Lower latency but can be more complex to implement.

Example

Using Apache Kafka and Apache Flink for real-time fraud detection in financial transactions.



Data processing – Stream processing example

Scenario

Real-time monitoring of credit card transactions to detect fraud.

Steps

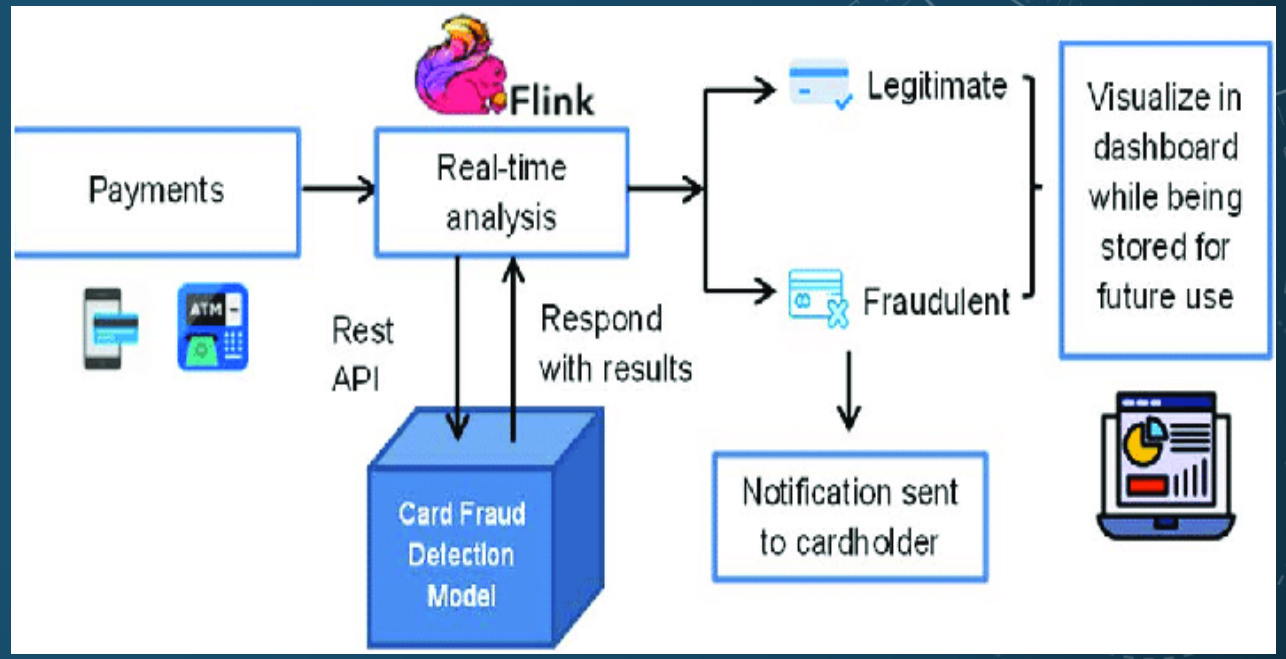
- Data Ingestion: Transactions are ingested from Kafka in real-time.
- Stream Processing: Apache Flink processes each transaction to identify anomalies.
- Anomaly Detection: Suspicious transactions are flagged immediately.
- Output: Real-time alerts are sent to the fraud detection team.

Advantages

Immediate detection and response to fraud.

Disadvantages

More complex to manage and scale.



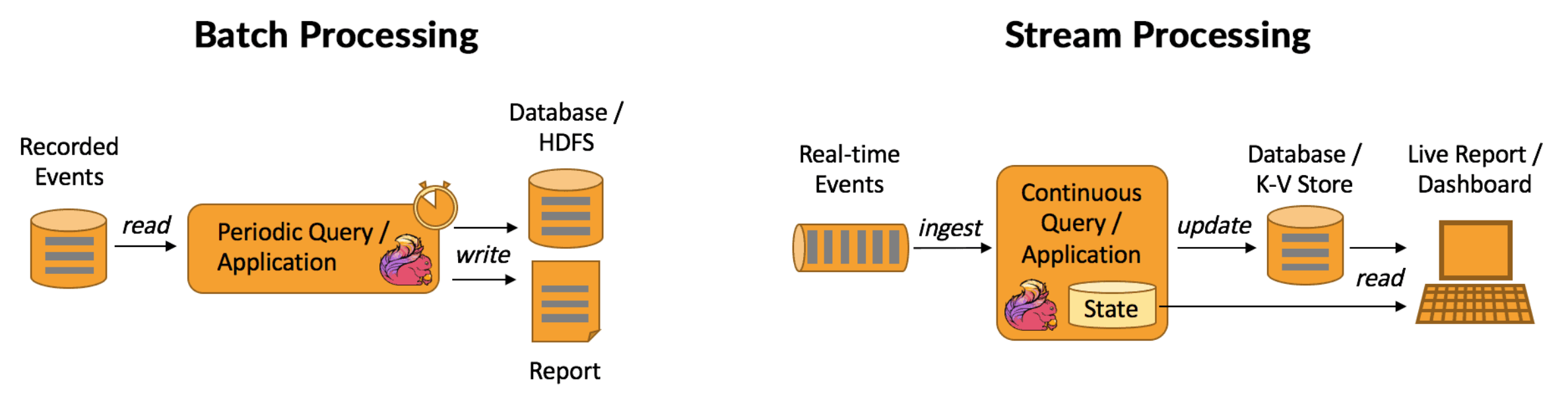
Data processing – Batch vs. Stream Processing

Comparison

Feature	Batch Processing	Stream Processing
Data Ingestion	Collected over time	Continuous
Processing Frequency	Scheduled intervals	Real-time
Latency	High	Low
Use Cases	Historical data analysis	Real-time monitoring
Complexity	Simpler	More complex

When to Use

- Batch Processing: When processing large volumes of historical data.
- Stream Processing: When immediate data processing and response are required.



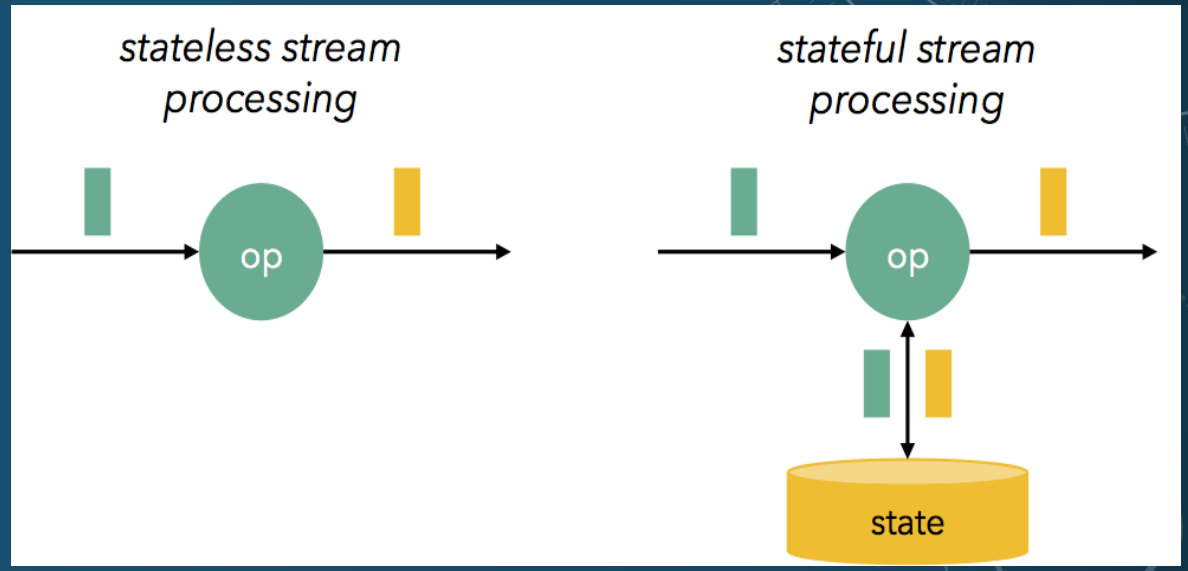
Data processing – Deep Dive into Stream Processing

Stateless Processing

Stateless streaming processing is a type of data processing where the system does not maintain any state or memory of past events. Each incoming data item is processed independently, without relying on information from previous items.

Common use cases for stateless streaming processing:

- Ingesting, cleaning, and transforming data.
- Filtering out unwanted data based on specific criteria.
- Applying transformations to data, such as converting data formats or calculating derived values.
- Calculating basic aggregations like sums, averages, or counts over a window of data.



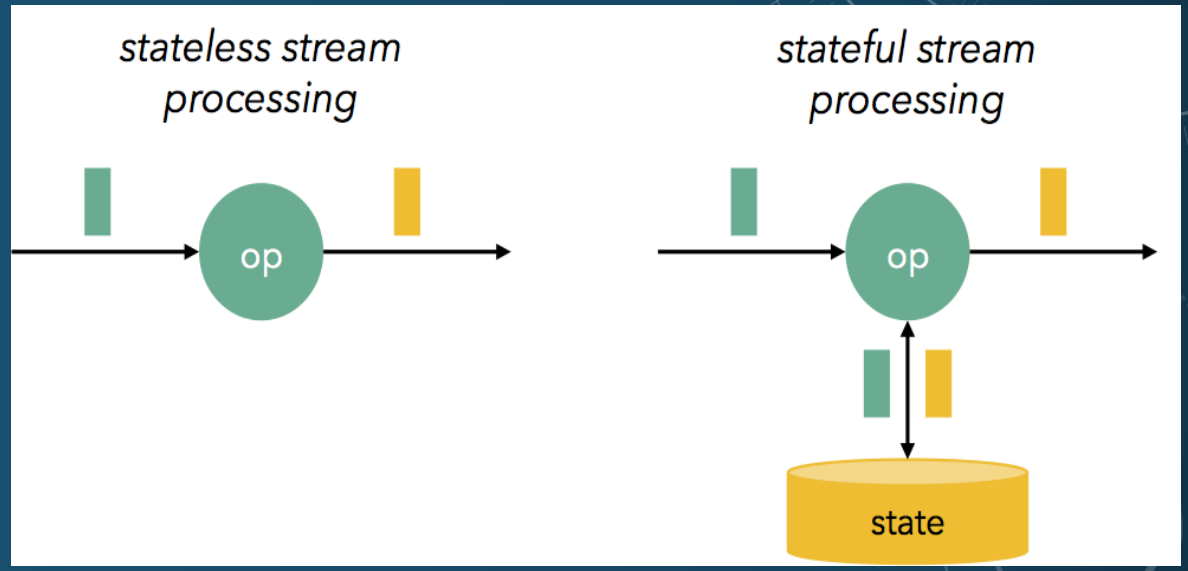
Data processing – Deep Dive into Stream Processing

Stateful Processing

Keeping track of state across multiple events.

Key concepts

- State: information about past events or calculations
- Stateful operators: maintain and update state. E.g.: *aggregate, reduce,* and *window*
- State store: storage mechanism for state. It can be in-memory, persistent storage (like HDFS or S3), or a combination of both.



Data processing – Deep Dive into Stream Processing

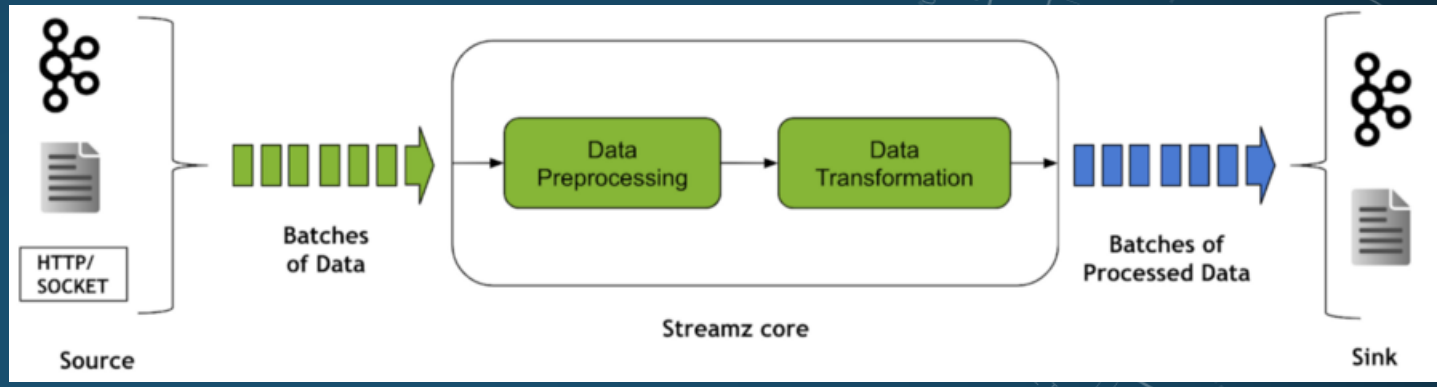
Types of time windows

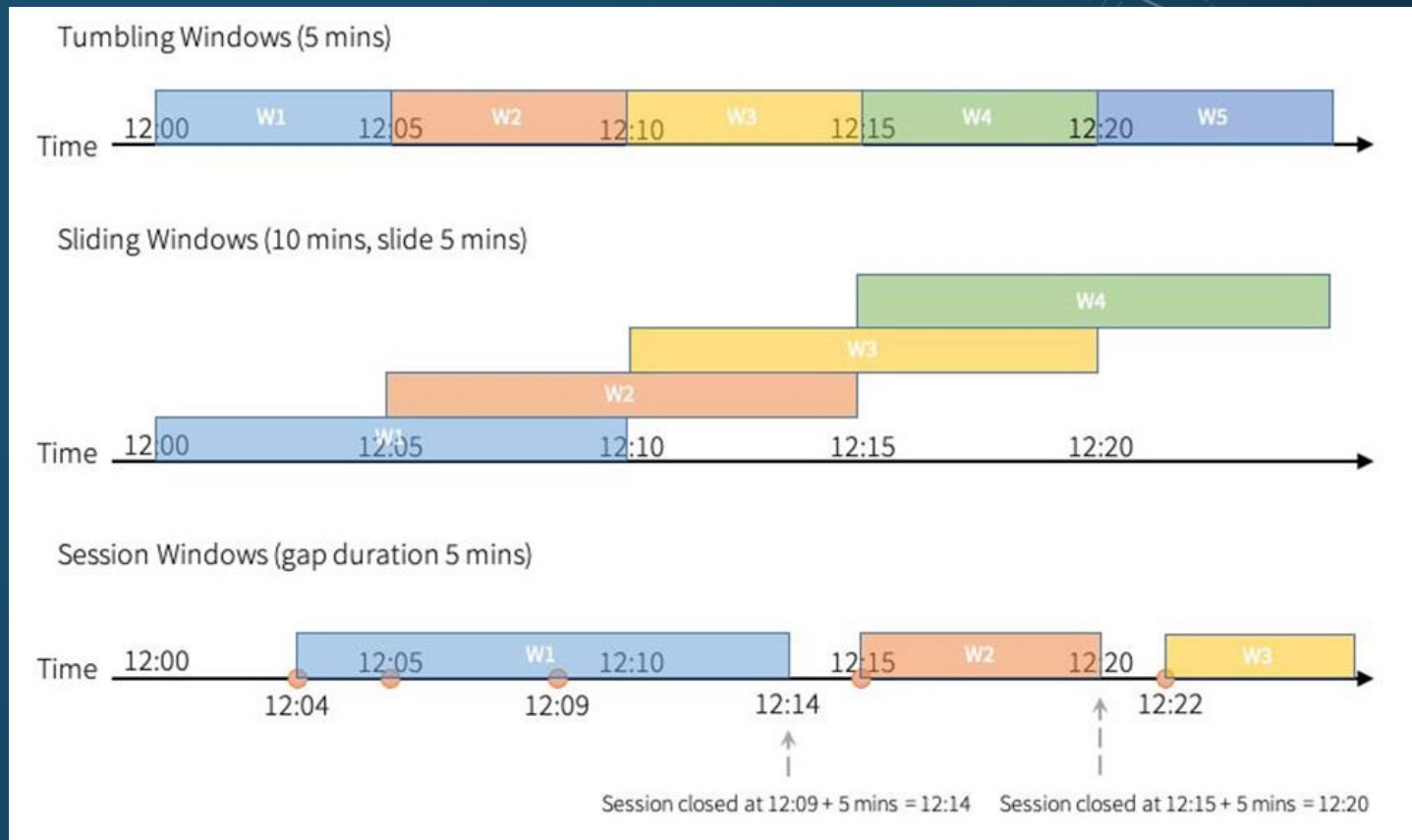
Windows

Time-based grouping of data streams

Types

- **Tumbling windows**: fixed-sized, non-overlapping, continuous time intervals; An input can only be bound to a single window.
- **Sliding windows**: fixed-sized, can overlap duration of slide is smaller than the duration of window. In this case an input can be bound to the multiple windows
- **Session windows**: dynamic size of the window length, depending on the inputs; starts with an input, and expands itself if following input has been received within gap duration. A session window closes when there's no input received within gap duration after receiving the latest input.





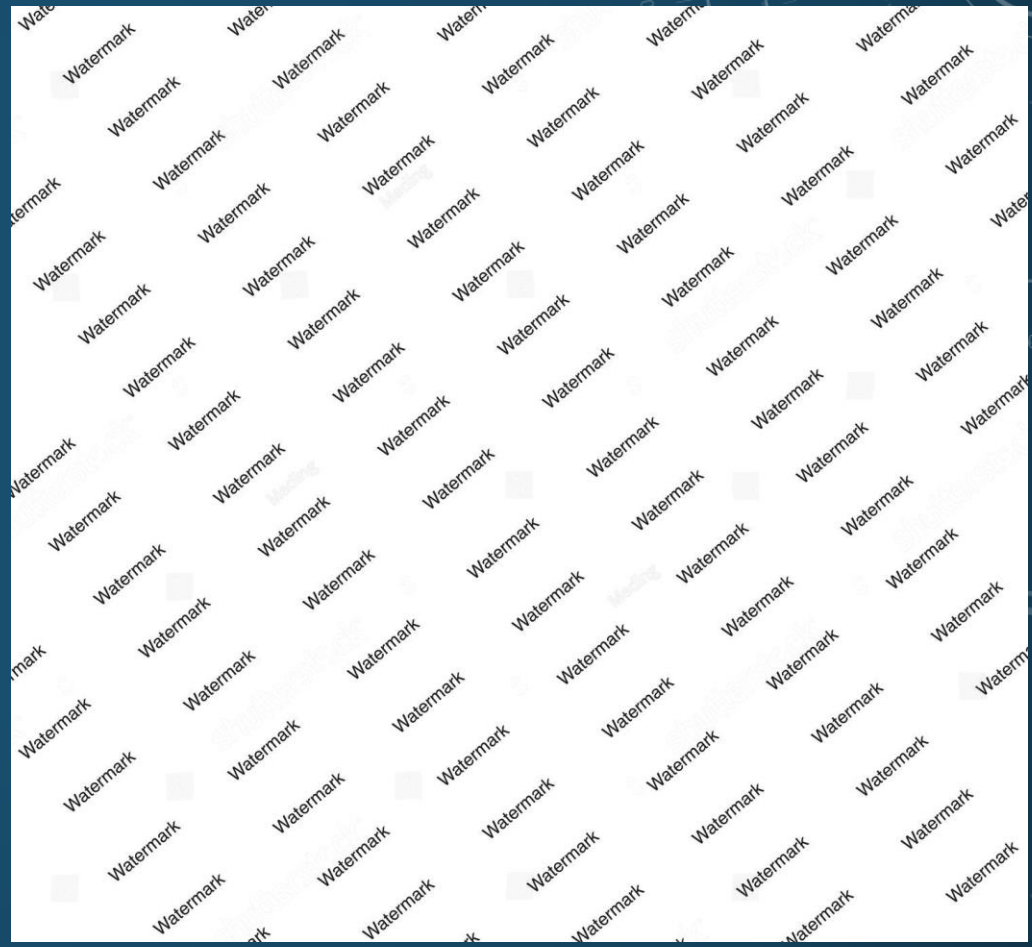
Data processing – Deep Dive into Stream Processing

Handle late events with watermark

Watermarks: a design or pattern embedded in paper during its manufacture.

However, in streaming processing, watermark is mechanism to handle late-arriving data.

The term "watermark" is a metaphor drawn from paper and printing. The watermark is a timestamp that's embedded in the stream of data.



Data processing – Deep Dive into Stream Processing

Handle late events with watermark

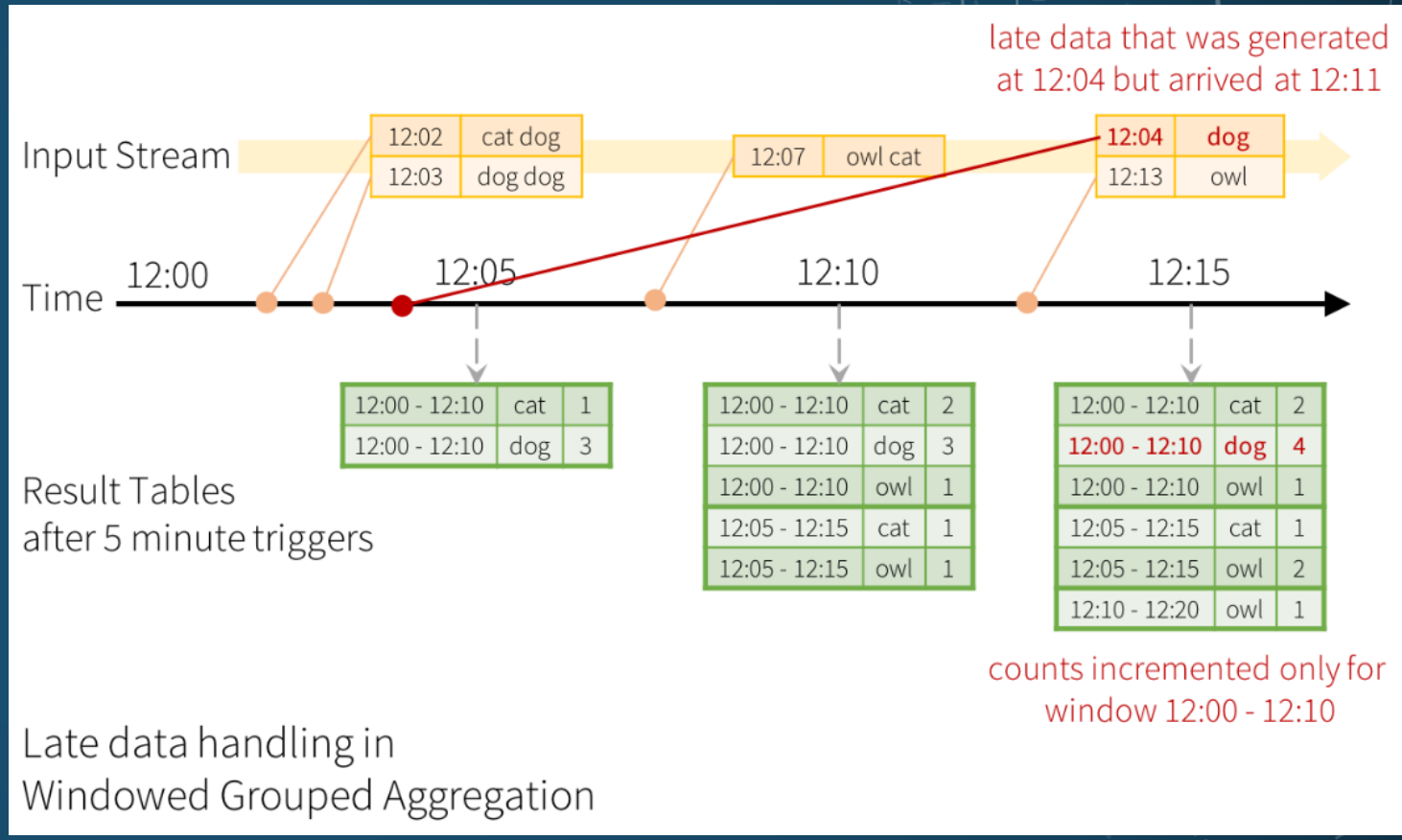
Watermarks: Mechanism to handle late-arriving data.

Types of timestamps:

- Event time: event occurred in the real world
- Arrived time / ingestion time
- Processing time
- Watermark timestamp: indicates the latest event time that has been processed by the system.

Late events handling:

- Buffering: store late events for later processing
- Processing with a delay: wait for a period, then process
- Dropping

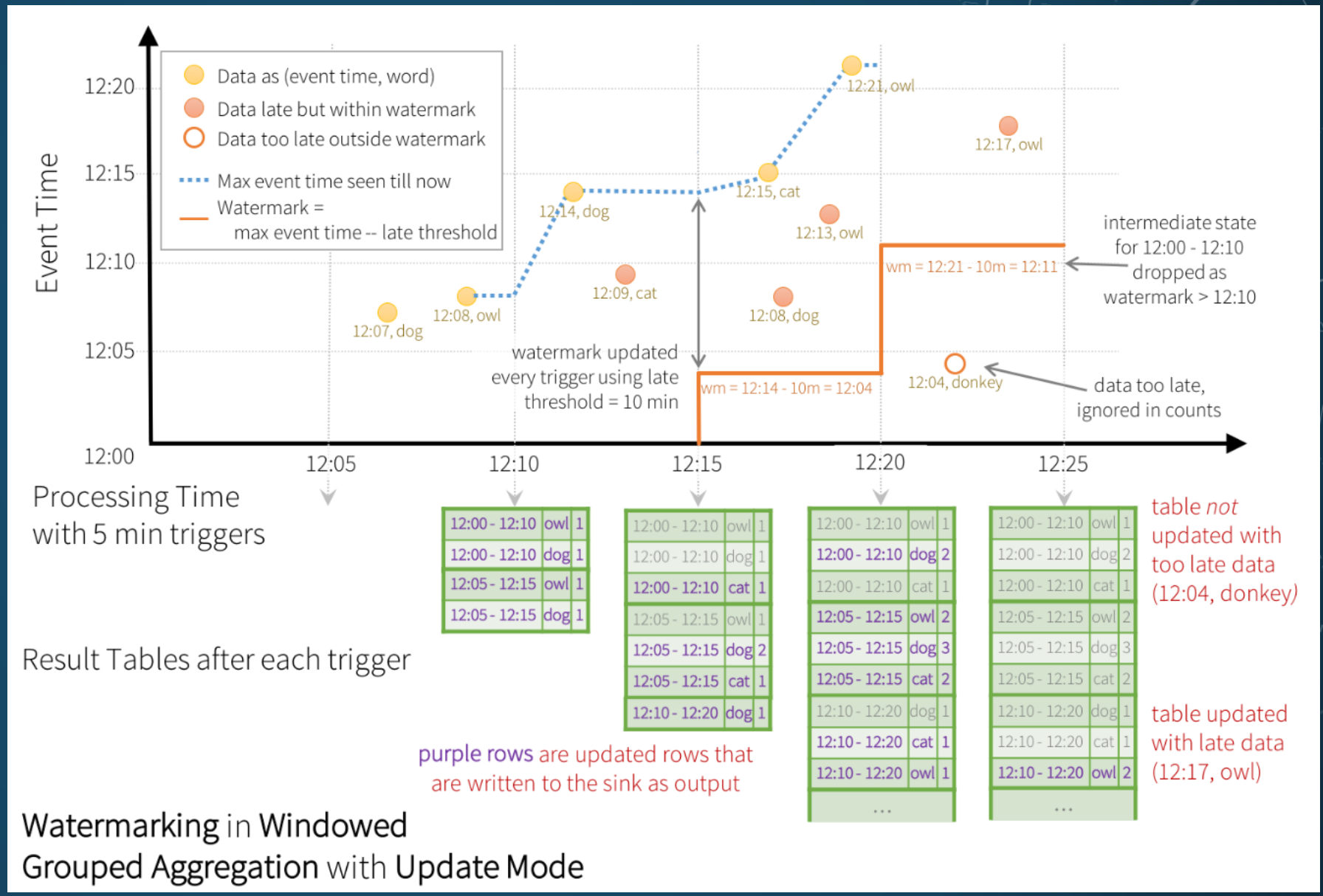


Data processing – Deep Dive into Stream Processing

Handle late events with watermark – example

Watermark column: “timestamp”

Threshold: 10 minutes



Data processing – Deep Dive into Stream Processing

Output modes

Append

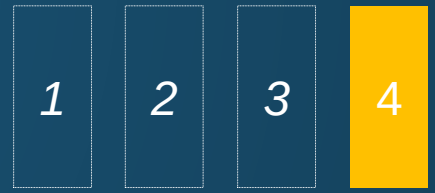
This mode appends new rows to the existing output data.

It's suitable for use cases where you want to accumulate data over time.

Complete

This mode overwrites the entire output dataset with each new batch of data.

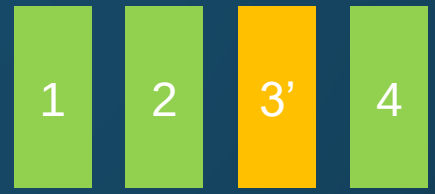
It's useful when you want to get the latest state of the data.



Update

This mode updates existing rows based on a key and updates the output dataset.

It's suitable for use cases where you want to modify existing data based on new information.



Data processing – Deep Dive into Stream Processing

Examples

Stateless:

Filtering financial transactions: Filtering out fraudulent transactions based on real-time rules.

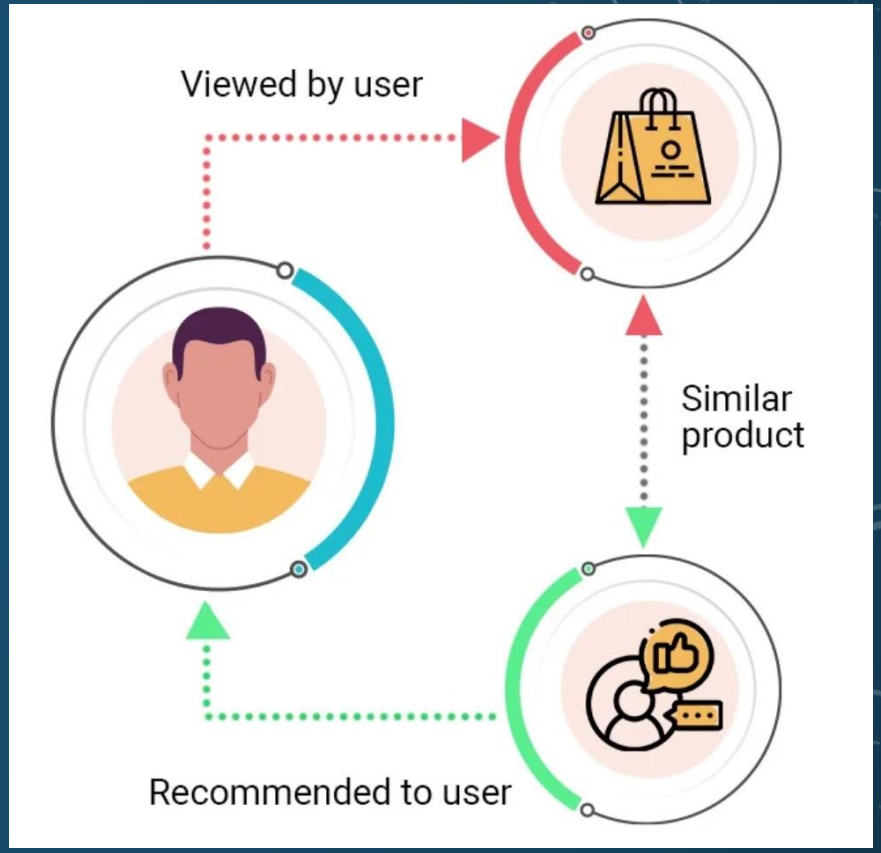
Analyzing IoT sensor data: Detecting anomalies in sensor data to identify equipment failures.

Stateful:

Recommender systems: Providing personalized recommendations based on user behavior and historical data.

Fraud detection: Detecting fraudulent activities by tracking user behavior over time and identifying suspicious patterns.

Inventory management: Tracking inventory levels in real-time and triggering replenishment orders when stock levels fall below a threshold.



DATA STORAGE

- Understand the architecture and functionality of Hadoop Distributed File System (HDFS).
- Explore the types and use cases of NoSQL databases in distributed systems.
- Learn the advantages and limitations of HDFS and NoSQL databases.
- Discuss real-world applications and examples.

Data storage in distributed systems

Distributed data storage refers to storing data across multiple machines to ensure:

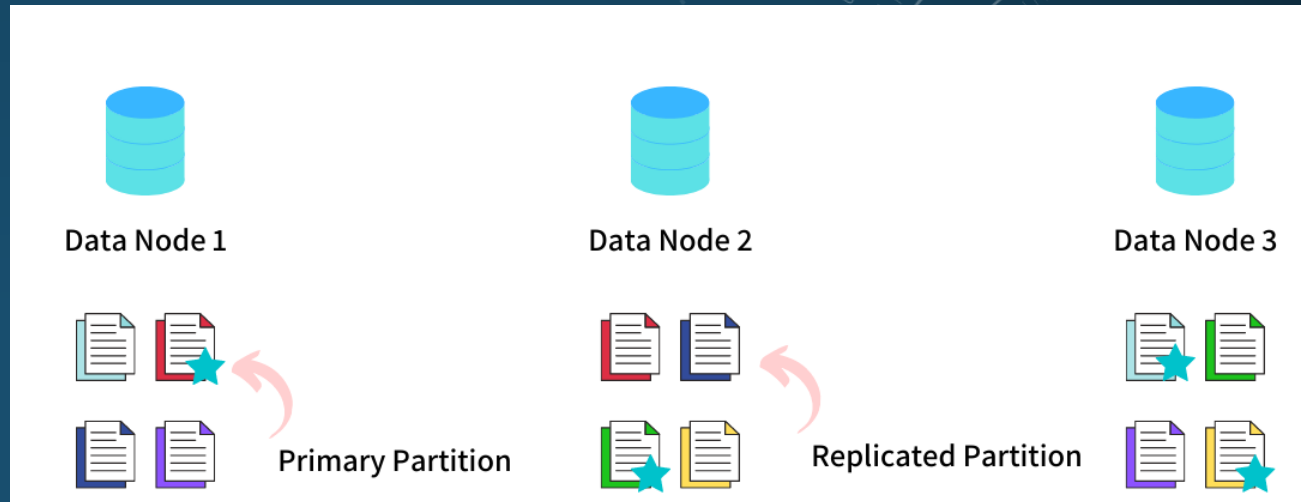
- Scalability
- Fault tolerance
- High availability (HA)
- Performance

Data storage - Data distribution

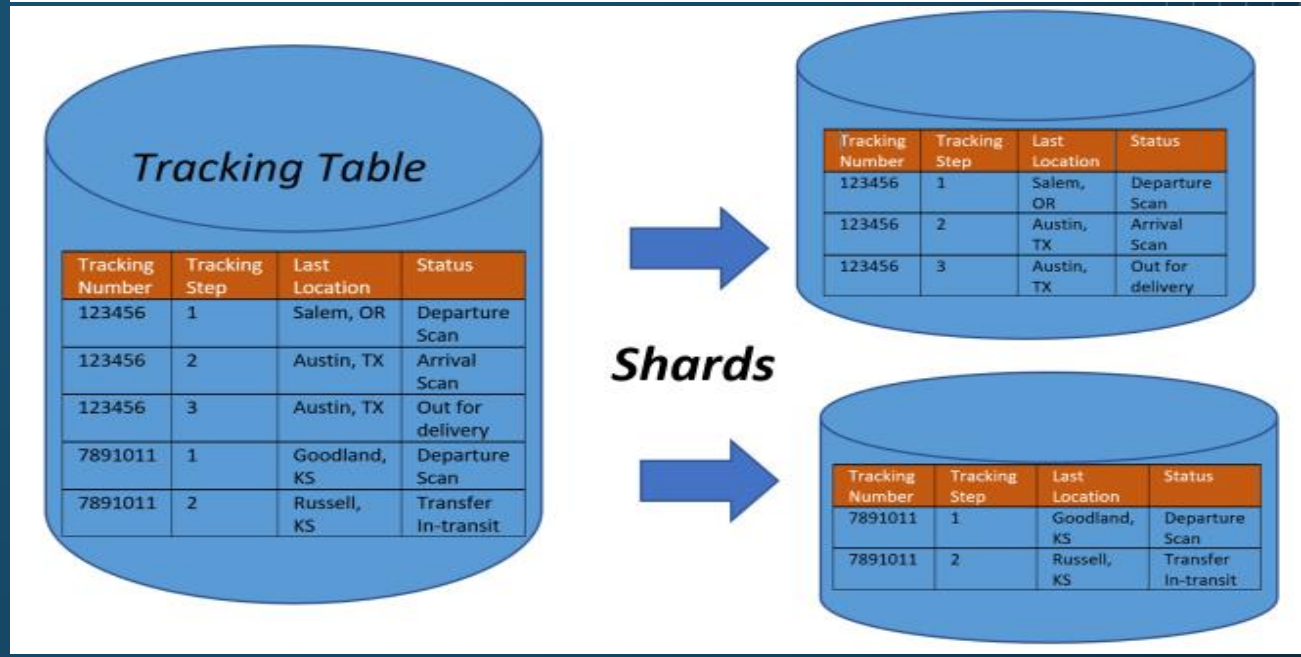
Partitioning: Dividing data into smaller chunks and assigning them to different nodes.

Replication: Copying data to multiple nodes for redundancy and fault tolerance.

Sharding: Distributing data across multiple databases based on a partitioning key.



Each data node holds the primary copy of one partition while the replicated copy is at another data node. So, if one node crashes we recover the data from another data node.



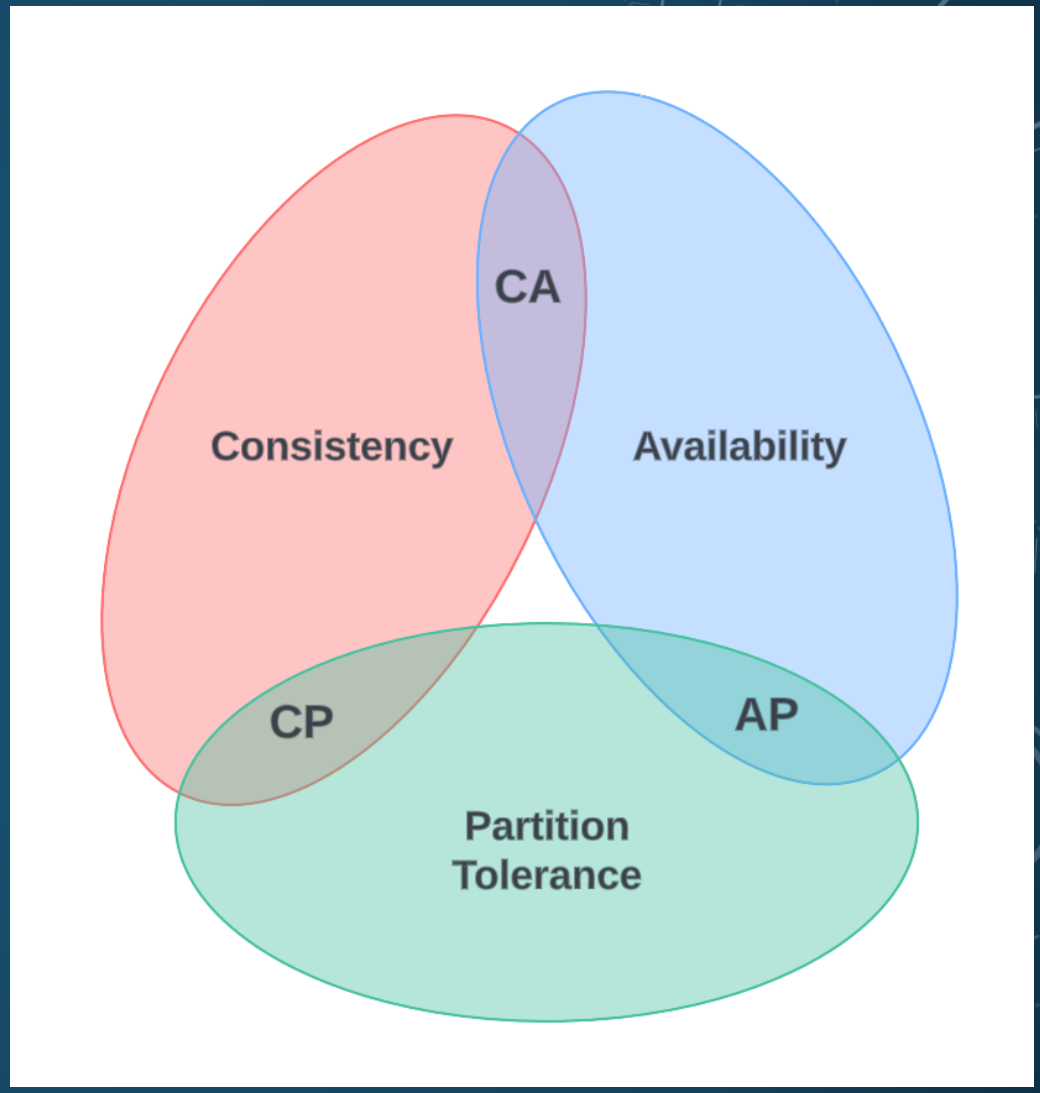
Data storage - Data consistency

Strong consistency: All copies of the data are always consistent.

Weak consistency: Eventually all copies of the data will be consistent.

Eventual consistency: Data will eventually be consistent across all replicas, but there may be temporary inconsistencies.

Take away: CAP theorem?

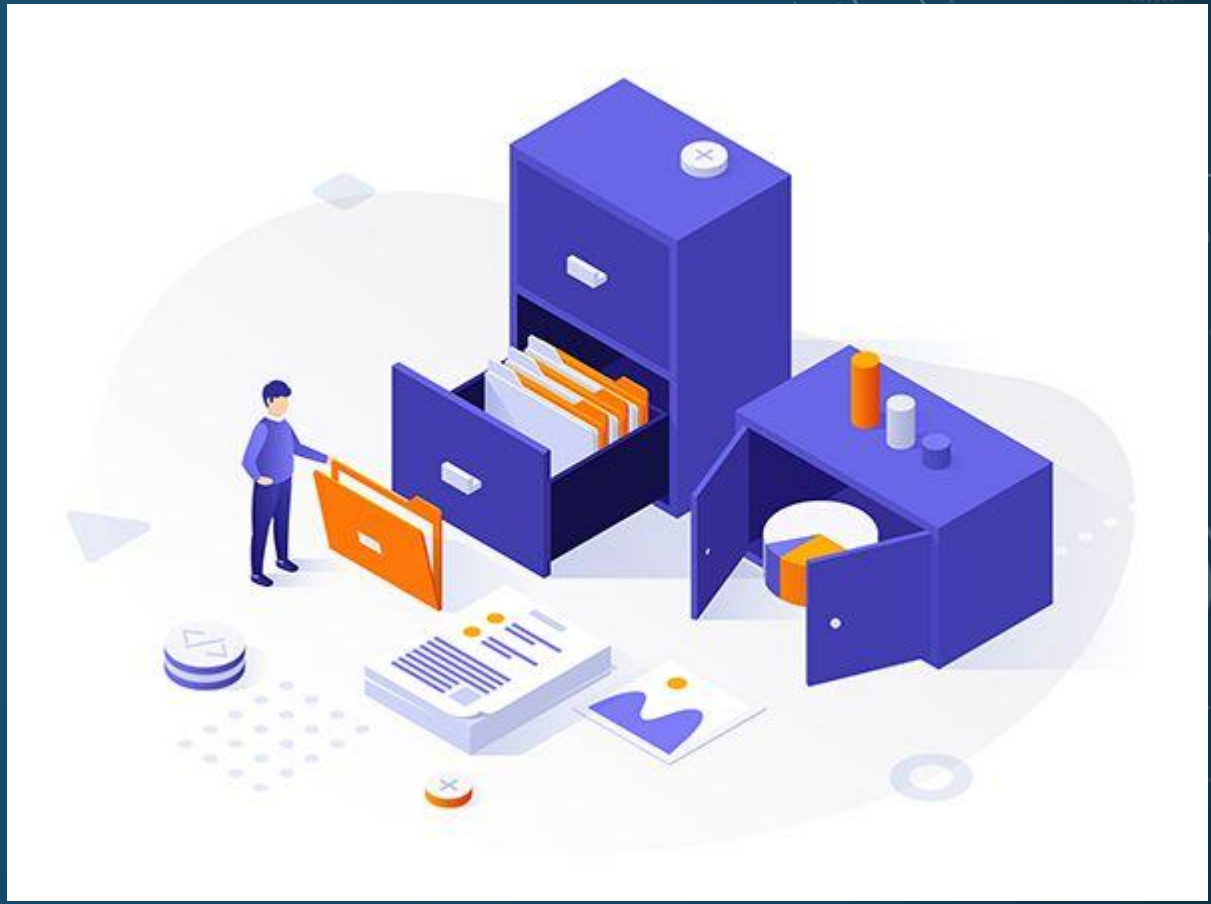


Data storage - Data management

Metadata: Information about data, such as location, format, and access control.

Indexing: Creating indexes to improve query performance.

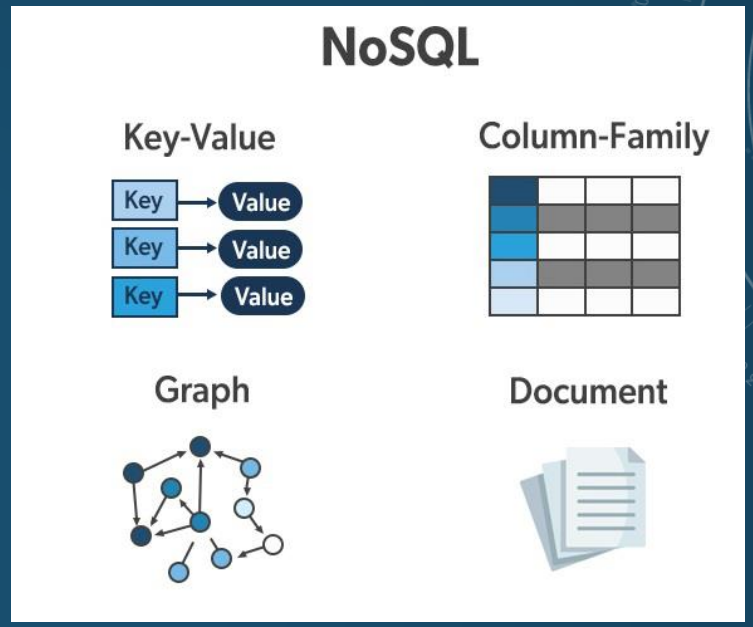
Query processing: Handling queries and returning results to clients.



Data storage – Common data storage models

NoSQL Databases

- Key-value stores: Store data as key-value pairs (e.g., Redis, Memcached).
- Document databases: Store data as documents (e.g., MongoDB, Couchbase).
- Wide-column databases: Store data as wide rows with multiple columns (e.g., Cassandra, HBase).



Distributed File Systems

Store data as files and directories (e.g., HDFS, Ceph).

Object Storage

Store data as objects (e.g., Amazon S3, Google Cloud Storage).

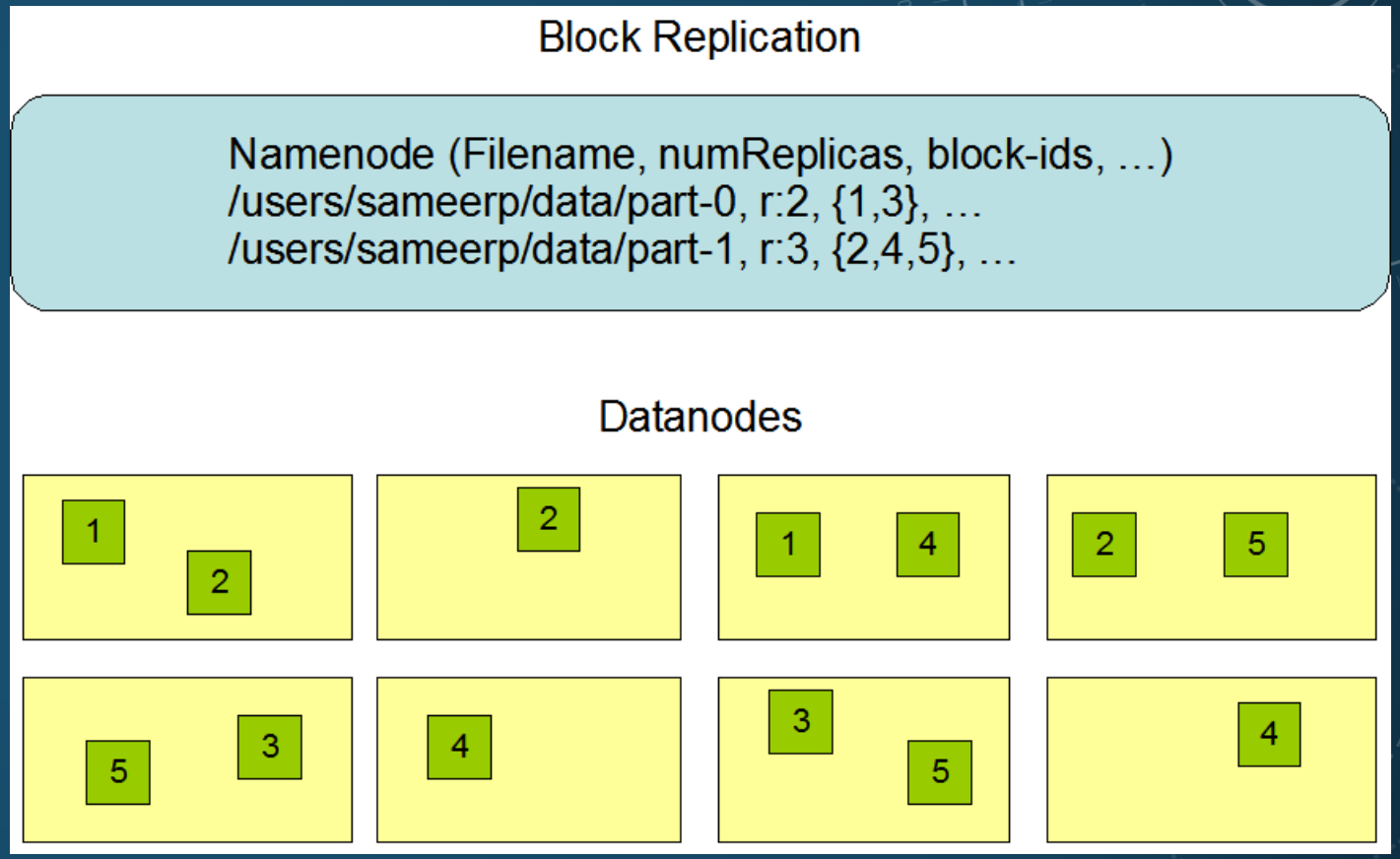
Data storage – Introduction to HDFS

HDFS is the primary storage system used by Hadoop applications. It is designed to store large datasets reliably and to stream those data sets at high bandwidth to user applications.

Data storage – Introduction to HDFS

Key Features:

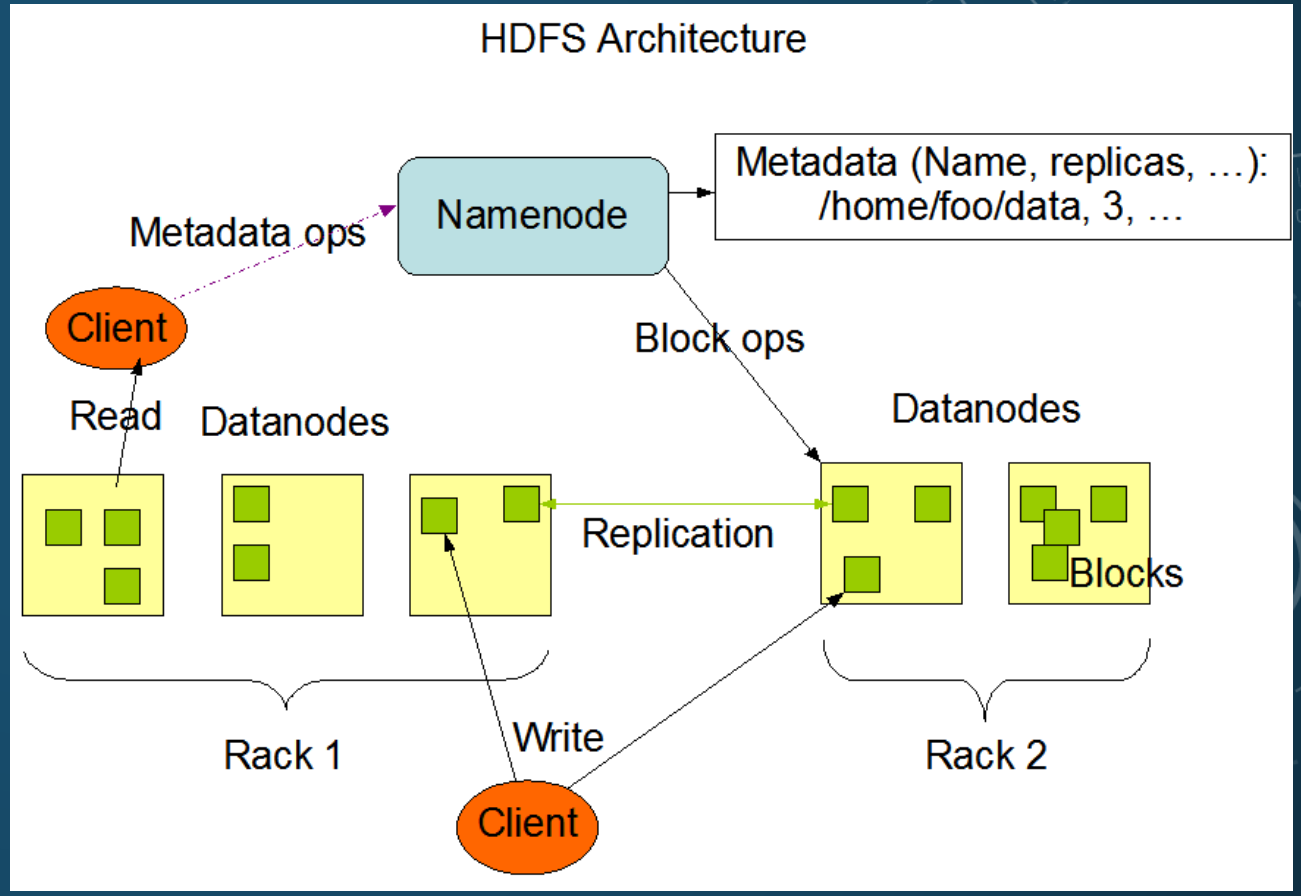
- Block Storage: Files are split into large blocks (typically 128MB) and distributed across the cluster.
- Replication: Blocks are replicated (usually 3 times) to ensure fault tolerance.



Data storage – Introduction to HDFS

Master-Slave Architecture:

- Namenode: Manages the metadata and directory structure of the file system.
- Datanodes: Store and manage the actual data blocks.



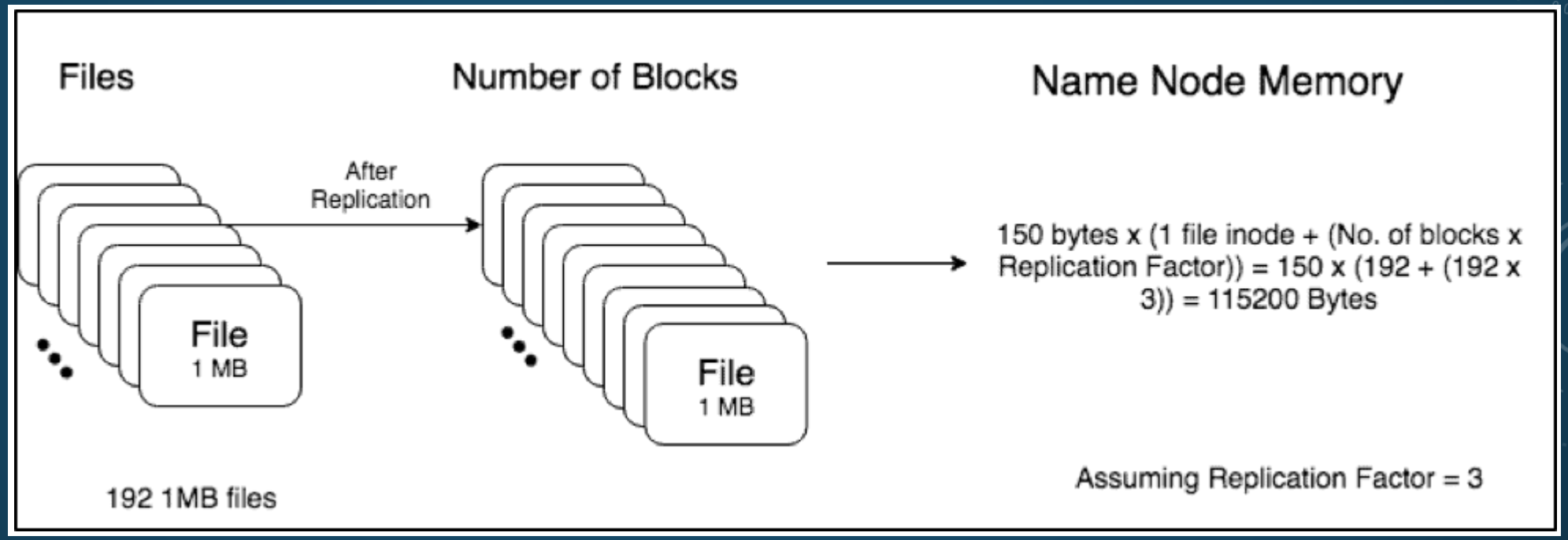
Data storage – Introduction to HDFS

Advantages:

- Scalability, fault tolerance, high throughput.

Disadvantages:

- Not ideal for small files, higher latency for low-latency needs.



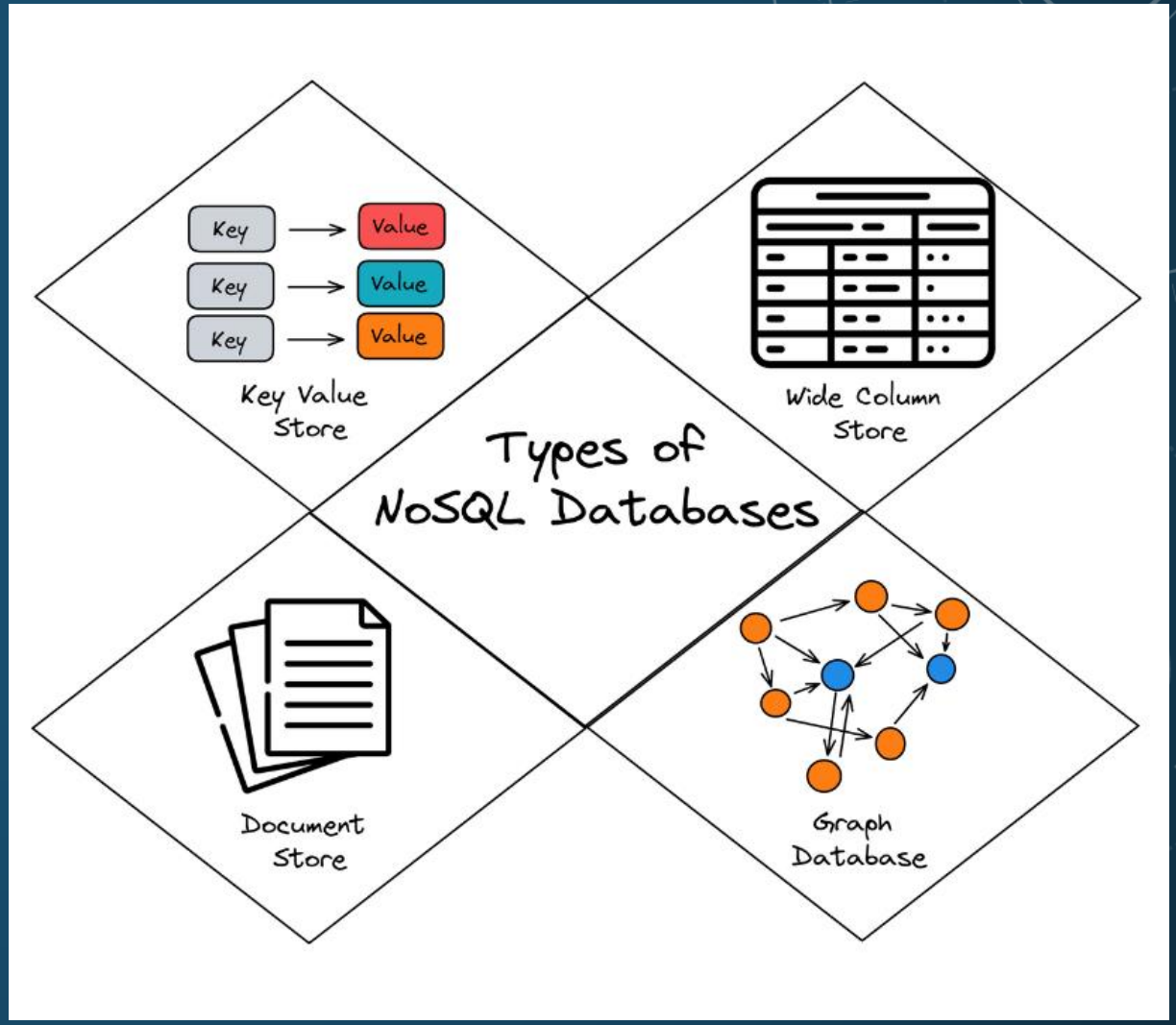
Data storage – Introduction to NoSQL Databases

NoSQL databases are designed to handle large volumes of unstructured or semi-structured data and are optimized for specific use cases like high availability, scalability, and flexibility.

Data storage – Introduction to NoSQL Databases

Types of NoSQL Databases:

- Key-Value Stores: e.g., Redis – Data is stored as key-value pairs.
- Document Stores: e.g., MongoDB – Stores data as JSON-like documents.
- Column-Family Stores: e.g., Apache Cassandra – Data is stored in columns rather than rows.
- Graph Databases: e.g., Neo4j – Optimized for storing and querying graph structures.



Data storage – Introduction to NoSQL Databases

Advantages:

Schema flexibility, scalability, and high performance for specific use cases.

Disadvantages:

Lack of ACID transactions in some NoSQL databases, complex consistency models.

Data storage – HDFS vs. NoSQL Databases

Feature	HDFS	NoSQL Databases
Data Structure	File-based (blocks)	Document, Key-Value, Column
Scalability	Horizontal scaling	Horizontal scaling
Fault Tolerance	Replication (3x default)	Varies by implementation
Schema	Fixed schema	Flexible schema
Use Cases	Batch processing, Big Data	Real-time applications, Unstructured data

When to use?

- **HDFS**: Best for large-scale batch processing tasks that require high throughput.
- **NoSQL**: Best for applications needing flexible schemas and real-time data access.