Distributed Storage & Data Processing

NDBI040 - Modern Database Systems

I. Holubová, P. Koupil

irena.holubova@matfyz.cuni.cz
pavel.koupil@matfyz.cuni.cz

https://www.ksi.mff.cuni.cz/~holubova/NDBI040/

Big Data Ecosystem – Technology Overview

- Distributed storage
  - File systems - HDFS
  - Object storage - Amazon S3, MinIO
- Distributed processing
  - MapReduce
  - Apache Spark
- Distributed databases
  - NoSQL, multi-model, NewSQL, …
- Search & indexing
  - Elasticsearch

Distributed Storage

- Why distributed storage?
  - Data volumes exceed capacity of a single machine
  - Need for scalability and fault tolerance
  - Data generated continuously (logs, sensors, web, media)
  - Processing must run close to data
- Key idea: data stored across many machines and accessed as a single logical system
- Core design principles
  - Horizontal scaling on commodity hardware
  - Failure is normal → replication and redundancy
  - High-throughput access to large files
  - Parallel access from many nodes

Apache Hadoop

- What is Apache Hadoop?
  - Open-source framework for distributed storage and processing of large data
  - Designed to run on clusters of commodity hardware
  - Inspired by Google File System and MapReduce
- Why Hadoop emerged
  - Rapid growth of web and log data
  - Limits of single-machine databases
  - Need for scalable and fault-tolerant infrastructure
- Core idea: Store and process large datasets across many machines as one system

http://hadoop.apache.org/

Main Hadoop Components

- HDFS → distributed storage layer
- YARN → resource management and scheduling
- MapReduce → distributed batch processing engine
- Hadoop Common → shared libraries and utilities

First widely adopted open-source Big Data platform and foundation of modern distributed data systems

Hadoop Ecosystem (Selected Projects)

- Storage & serving
  - HBase — column-family distributed database
- SQL & analytics
  - Hive — SQL-like analytics over HDFS
- Coordination
  - ZooKeeper — distributed coordination service
- Serialization / data format
  - Avro — schema-based data serialization

Hadoop evolved into a large ecosystem of specialized tools for storage, processing and analytics

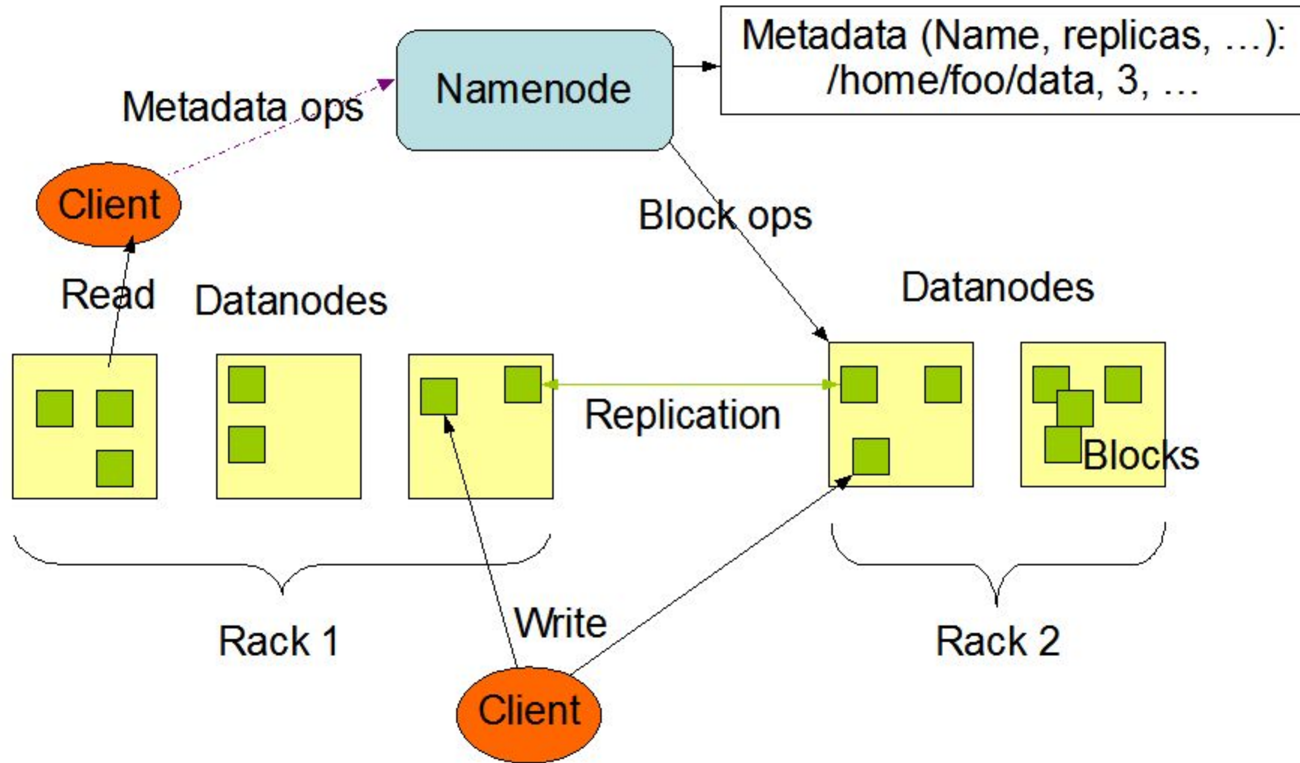
HDFS (Hadoop Distributed File System)

- Abstraction - distributed file system
  - Behaves like a standard file system
- Namespace
  - Hierarchical structure (directories + files)
  - Paths (e.g., `/data/logs/file.txt`)
- Typical operations
  - `mkdir, ls, put / get, mv, rm`
- Access methods
  - Command line (`hdfs dfs …` / `hadoop fs …`)
  - Java API (and other language bindings)
  - Web interface (for browsing & monitoring)

Same interface as a file system — different internal implementation

HDFS - Architecture

- Storage layer of the Hadoop ecosystem
- Master-slave model
  - NameNode → metadata management
  - DataNodes → store actual data blocks
- Data model
  - Files split into large blocks (128 MB+)
  - Blocks distributed across cluster nodes
  - Each block replicated (default: 3 copies)


HDFS Architecture
Metadata (Name, replicas, …):
/home/foo/data, 3, …
Namenode
Metadata ops
Client
Block ops
Read
Datanodes
Datanodes
Replication
Blocks
Rack 1
Write
Rack 2
Client

HDFS - Fault Tolerance

- Design philosophy: “Failure is the norm, not the exception.”
- Why?
  - Clusters = thousands of machines
  - Hardware failures are statistically inevitable
- System assumption
  - At any time, some component is failing
- System response
  - Fault detection
  - Automatic re-replication
  - Background recovery

HDFS - Data Characteristics

- Access pattern
  - Streaming data access
  - High throughput preferred
- Update model
  - Files typically written once
  - Appends possible, random updates rare
- Typical usage
  - Log data
  - Web-scale datasets
  - Analytical processing inputs
- Implication: Optimized for large-scale analytics, not interactive data access

HDFS - How NameNode Works?

- Metadata in memory
  - NameNode stores file system metadata in memory
  - Namespace: files, directories, permissions
  - Mapping of files to blocks and DataNodes
- Persistent metadata on disk
  - **FsImage** → snapshot of file system metadata
  - **EditLog** → record of all metadata changes
- Startup & recovery
  - On startup: FsImage loaded into memory
  - EditLog replayed to reconstruct latest state
- Key idea: Separation of metadata snapshot and change log ensures reliability and recovery

HDFS - How DataNode Works?

- Storage role
  - Stores actual data blocks on local disks
  - Each block stored as a local file
  - One DataNode typically per cluster node
- Responsibilities
  - Serves read/write requests from clients
  - Creates, deletes and replicates blocks
  - Executes instructions from NameNode
- Cluster communication
  - Sends periodic **heartbeats** to NameNode
  - Reports stored blocks

Processing Distributed Data

- Now we can store massive data across many machines
  - But how can we process them?
- Challenges
  - Data distributed across cluster
    - Too large for a single machine
  - Moving data over network is expensive
  - Failures during computation are common
- Need: A programming model for distributed data processing

Google

MapReduce – Basic Idea

- A programming model and execution framework
- Origin: Google (Dean & Ghemawat, 2004)
  - Designed for processing web-scale data
  - Inspired Hadoop MapReduce and modern data engines
- Core principle: **divide and conquer**
  - Split data into parts
  - Process parts in parallel
  - Combine partial results
- Framework handles
  - Data distribution
  - Parallel execution
  - Fault tolerance
- Programmer defines only the computation logic

MapReduce – Functions Map and Reduce

- Function **Map** $(k_1, v_1) \rightarrow list(k_2, v_2)$
  - Processes input data
  - Produces intermediate key–value pairs
  - Runs in parallel on many nodes
- Function **Reduce** $(k_2, list(v_2)) \rightarrow (k_2, possibly\ smaller\ list(v_2))$
  - Receives all values with the same key
  - Aggregates or combines them
  - Produces final results

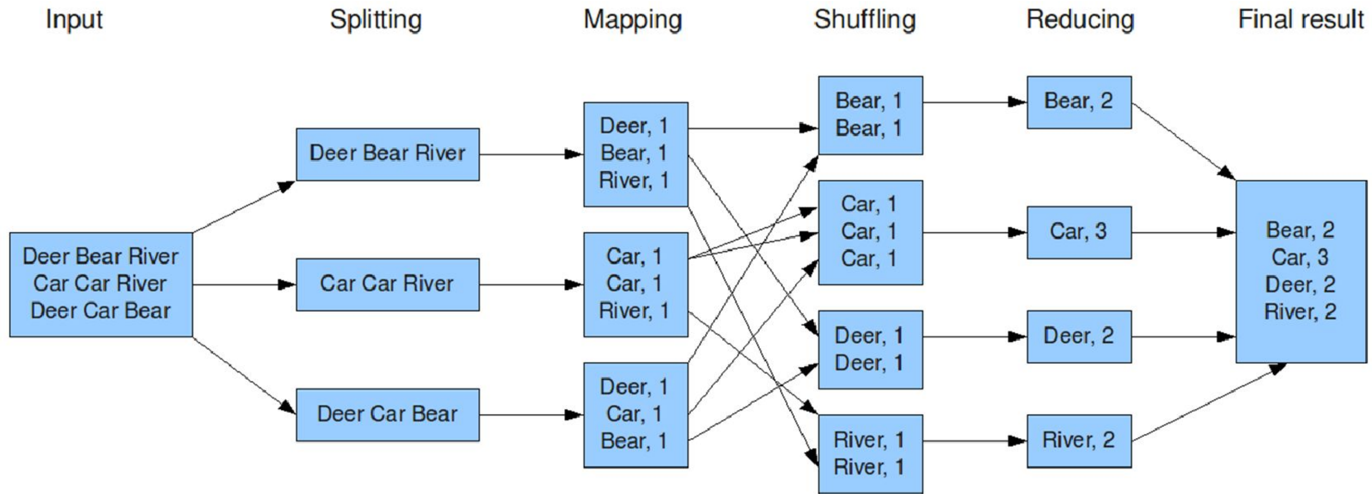
Key idea: Map transforms data ⟶ Reduce aggregates results

Example: Word Frequency

```
map(String key, String value):
  // key: document name
  // value: document contents
for each word w in value:
  EmitIntermediate(w, "1");
```

```
reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
int result = 0;
for each v in values:
  result += ParseInt(v);
Emit(key, AsString(result));
```

Example: Word Frequency



Why MapReduce Was Revolutionary

Before MapReduce:

- Distributed computing was complex
- Manual parallelization
- Manual fault tolerance

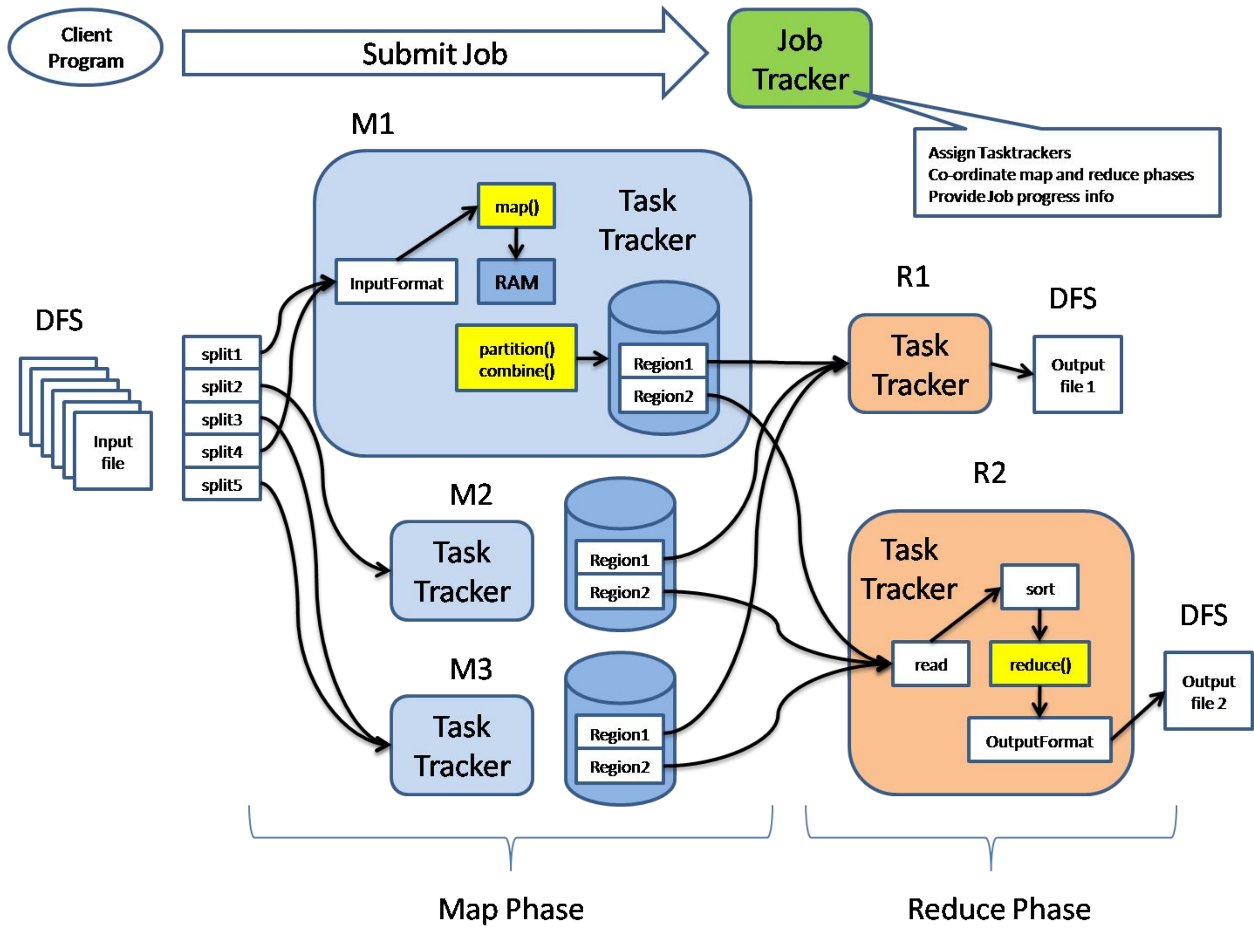
MapReduce provided:

- Simple programming model
- Automatic parallelization
- Built-in fault tolerance
- Scalable processing on clusters

Hadoop MapReduce

- Open-source implementation of the MapReduce model within the Hadoop ecosystem
- Role
  - Distributed batch processing engine
  - Runs on top of HDFS
- Characteristics
  - Processes large datasets
  - Disk-based intermediate storage
  - Designed for high-throughput batch jobs
- JobTracker – master node
  - Schedules jobs and tasks, monitors execution
- TaskTracker – worker nodes
  - Execute map and reduce tasks, report status to JobTracker

Historical importance: First widely adopted Big Data processing framework



Client Program
Submit Job
Job Tracker
Assign Tasktrackers
Co-ordinate map and reduce phases
Provide Job progress info
M1
Task Tracker
map()
InputFormat
RAM
partition()
combine()
Region1
Region2
DFS
Input file
split1
split2
split3
split4
split5
R1
Task Tracker
DFS
Output file 1
R2
Task Tracker
sort
read
reduce()
OutputFormat
DFS
Output file 2
M2
Task Tracker
Region1
Region2
M3
Task Tracker
Region1
Region2
Map Phase
Reduce Phase

Hadoop MapReduce – Execution Roles

- **Mapper**
  - Processes input
  - Produces intermediate key–value pairs
  - Runs in parallel across cluster
- **Partitioner**
  - Determines which reducer receives each key
  - Ensures same keys go to the same reducer!
  - Default: hash(key)
- **Reducer**
  - Receives grouped values for each key
  - Aggregates or combines them
  - Produces final output
- Combiner (optional)
  - Local mini-reducer on mapper node
  - Reduces amount of data sent during shuffle
- Stragglers
  - Slow tasks that delay job completion
  - Re-execution of slow tasks

MapReduce – Criticism

- David DeWitt & Michael Stonebraker (2008)
  - Database community critique of MapReduce
- Main arguments
  - Step backwards from declarative query languages (SQL)
  - Lacks schema and data independence
  - Uses brute-force scans instead of indexes
  - Missing DBMS features
    - indexes
    - transactions
    - query optimization

Key message: MapReduce is powerful for large-scale batch processing, but not a replacement for database systems.

Note: Who is Michael Stonebraker (*1943)?

- Computer scientist and database researcher
  - Professor at UC Berkeley and MIT
- Turing Award recipient (2014)
  - “Nobel Prize of computing”
  - For concepts and practices underlying modern database systems
  - 2016 – Sir Tim Berners Lee
    - For inventing the WWW
- Major contributions
  - Ingres (relational DBMS)
  - Postgres → PostgreSQL
  - Vertica (column-oriented DBMS)

End of MapReduce?

- MapReduce was a breakthrough for large-scale distributed processing
- But… limitations emerged
  - High latency (disk-based processing)
  - Rigid two-stage model (map → reduce)
  - Inefficient for iterative algorithms
  - Poor support for interactive analytics
- Industry shift
  - Move toward faster, more flexible engines
  - In-memory processing
  - Real-time and interactive analytics

Result: New generation of distributed processing systems

From MapReduce to Apache Spark

- MapReduce enabled large-scale distributed processing
- But modern data workloads required more flexibility and speed
- New requirements
  - Faster processing (lower latency)
  - Interactive data analysis
  - Iterative algorithms (machine learning)
  - Unified support for batch and streaming

- Key idea: Keep data in memory and support general execution graphs

- Apache Spark – a unified engine for distributed data processing

Separation of Compute & Storage

- Traditional Hadoop model:
  - Storage (HDFS) and compute tightly coupled
  - Data stored locally on cluster nodes
    - We cannot scale storage and compute separately
- Modern architecture:
  - Storage independent of compute
  - Object storage (S3, MinIO)
  - Compute clusters scale independently
- Benefits
  - Elastic scaling
  - Better resource utilization
  - Cloud-native design
- Key idea: Compute and storage become separate layers

Data live in storage, compute comes and goes

From Distributed File Systems to Object Storage

- Separating compute and storage → a storage layer that is:
  - scalable
  - shared by many compute clusters
  - independent of specific machines
- Traditional HDFS:
  - tied to compute cluster
  - hard to scale independently
  - difficult to share across systems
- Modern solution: **object storage**
  - Shared persistent storage
  - Access via network API
  - Independent scaling
  - Cloud-native architecture
- Used by: Spark, analytics systems, machine learning, …

Amazon S3 Model

Amazon S3

- S3 (Simple Storage Service) = scalable distributed storage layer
  - Cloud service
  - Not a database (no SQL, indexes, transactions. constraints…)
- Data model:
  - **Bucket** (container, namespace)
  - **Object** = key (like path inside a bucket) + data (bytes) + metadata (attributes)
  - **Logical prefixes** (like folders, but not real)
    - The storage is flat
- API operations:
  - `PUT, GET, LIST, DELETE`
- Objects are immutable
  - Updating = overwrite whole object
  - No in-place updates

https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html

Data Layout in Object Storage

Amazon S3

- Objects are flat → structure is logical (prefix-based)
- Example:
  - `data/raw/transactions.csv`
  - `data/processed/daily_revenue/`
  - `data/analytics/`
- Partitioning via prefixes:
  - `data/processed/region=EU/date=2024-01-01/`
- Key idea:
  - Layout = query optimization
  - No schema → design in paths

MinIO

- High-performance object storage compatible with the Amazon S3 API
  - Stores data as objects in buckets
  - Accessible over network via HTTP API
  - Can run locally or in cloud
- Why we use it?
  - Fully S3-compatible
  - Lightweight and easy to run
  - Suitable for experiments and development
  - Same API as real cloud object storage
- How it is accessed
  - Web interface (browser)
  - Command line client (`mc`)
  - Python (`boto3`)
  - Apache Spark (`s3a:// connector`)

https://www.min.io/

MinIO - Basic Operations

- Upload object:
  `PUT data/raw/users.csv`

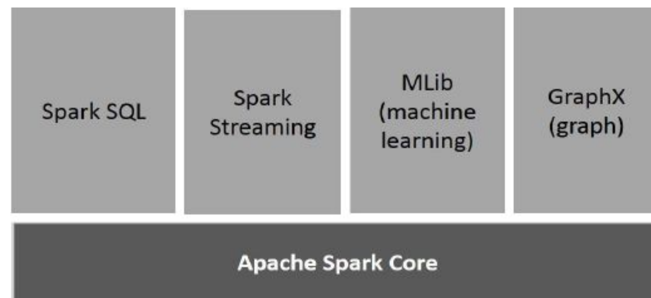
- Download object:
  `GET data/raw/users.csv`

- List objects:
  `LIST data/raw/`

- Delete object:
  `DELETE data/raw/users.csv`

Apache Spark (2014)

- Unified platform for large-scale data processing across distributed clusters
- Provides
  - Batch processing
  - SQL analytics (Spark SQL)
  - Machine learning (MLib)
  - Graph processing (GraphX)
  - Stream processing (Spark Streaming)

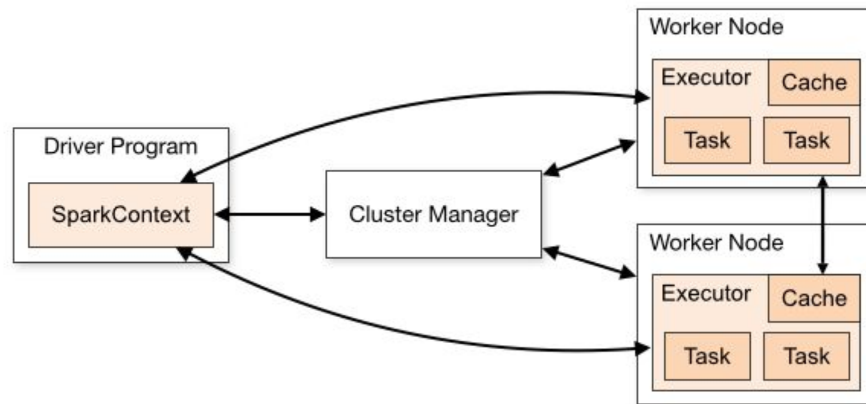


Key feature: General distributed execution engine for diverse data workloads

MapReduce vs Apache Spark

Aspect	MapReduce	Apache Spark
Processing model	Rigid map → reduce pipeline	General DAG execution engine
Intermediate data	Written to disk	In-memory when possible
Performance	Higher latency	Faster, low-latency processing
Workloads	Batch processing	Batch, SQL, streaming, ML, graph
Interactivity	Not interactive	Supports interactive analytics
Flexibility	Limited programming model	Flexible distributed computations
Typical role today	First-generation batch engine	Unified modern data processing engine

Spark Application



- Driver Program
  - Runs user’s main function
  - Creates SparkSession / SparkContext
  - Builds execution plan
  - Sends tasks to cluster
- Executors (on worker nodes)
  - Run tasks in parallel
  - Store data in memory or disk
  - Return results to driver
- Cluster Manager
  - Allocates resources
  - Launches executors

Driver → Executors → Tasks → Results

Spark Application

```
from pyspark.sql import SparkSession

spark = SparkSession.builder \
.appName("MyApp") \
.getOrCreate()
df = spark.read.csv("data.csv", header=True)
```

- **Spark session**
  - Entry point to Spark functionality.
  - Creates connection to cluster and initializes execution environment.
- Typical steps
  - Create SparkSession
  - Load data
  - Apply transformations
  - Execute actions
- In practice
  - Most work done via DataFrame / Spark SQL API

Resilient Distributed Dataset (RDD)

- Basic distributed data structure in Spark
  - Collection of elements partitioned across cluster
  - Processed in parallel
- Properties
  - Immutable
  - Fault-tolerant
  - Can be cached in memory
- Role today
  - Low-level API
  - Foundation of Spark execution engine

DataFrame

- Distributed table with schema
- Similar to:
  - relational table
  - pandas DataFrame
- Features
  - Structured data
  - Optimized execution
  - SQL support

- Role today
  - Main abstraction in modern Spark

Operations

Transformations

- Create a new dataset from an existing one
  - Lazily
- Build execution plan
- Do NOT run immediately

```
df.select("name")
df.filter(df.age > 18)
df.groupBy("age").count()
```

Actions

- Trigger execution and return results.
- Execute the computation
- Return result to driver

```
df.show()
df.count()
df.collect()
```

Important: Nothing happens until an action is called

Transformations

- Basic transformations
  - `map()` – apply given function to each element
  - `filter()` – keep only selected elements
  - `union()` – combine given datasets
  - `distinct()` – remove duplicates
- Key-based transformations
  - `reduceByKey()` – aggregate values by key
  - `groupByKey()` – group values by key
  - `sortByKey()` – sort by key

Important: Executed only when an action is called

Actions

- Return values to the driver
  - `count()` – number of rows
  - `collect()` – return all data
  - `first()` / `take(n)` – preview data
- Display results
  - `show()` – print table
- Write output
  - `write()` – save to file (CSV, JSON, Parquet, …)

Important:

- Actions start execution
- All previous transformations are executed together
- Result is either returned or written

Execution Hierarchy

- Action starts a **job**
- Job is divided into **stages**
- Stages consist of parallel **tasks**
  - Shuffle usually creates stage boundaries

Distributed Output

- Spark writes datasets as directories of part files
- Not as one monolithic output file
- Reflects parallel execution
- Post-processing may be needed for single-file export

Shuffle Operations

- Some operations require data redistribution
- Examples
  - `groupBy()`
  - `join()`
  - `reduceByKey()`
  - `orderBy()`
- What happens?
  - Data move across executors
  - Network + disk I/O
  - Expensive operation

Key idea: Shuffle is the main cost in distributed data processing

Closures

```
threshold = 18
df.filter(df.age > threshold).show()
```

- When Spark executes operations on worker nodes, it sends the function and referenced variables
  - Each worker has its own copy
  - This is called a **closure**
- Changes inside tasks do NOT modify driver variables
  - Key idea: Code must be self-contained for distributed execution

Accumulators

- Variables used for aggregated updates from worker nodes to the driver
- Properties
  - Workers can update
  - Only driver can read final value
  - Used for counters, statistics, debugging

```
counter = spark.sparkContext.accumulator(0)

def f(row):
    global counter
    counter += 1

df.foreach(f)
```

Broadcast Variables

Without broadcast, every chef receives their own copy of the recipe book for each dish they prepare. With broadcast, each kitchen gets one shared recipe book that all chefs reuse.

- Normally:
  - Variables are sent to workers with each task
- With broadcast:
  - Data are sent once to each worker and reused by all tasks
- Used for large read-only data

```
rates = {"CZ": 25.0, "EUR": 1.0}
rates_b = spark.sparkContext.broadcast(rates)

def convert(row):
    return row.country, rates_b.value[row.country]

df.rdd.map(convert).collect()
```

Spark SQL

Spark SQL

- Module for processing structured data in Spark.
  - Abstraction: **DataFrame** = distributed table with schema
- Provides
  - SQL queries
  - DataFrame API
  - Integration with Python
- Why important
  - Familiar relational model
  - Declarative queries
  - Scalable processing of large datasets

Key idea: Spark becomes a distributed SQL engine

File Formats for Analytics

- **CSV**:
  - row-based
  - human-readable
  - slow for analytics
  - Example: exported transactions, logs, data exchange

```
tr_id,timestamp,user_id,region,product_cat,amount
1,2024-01-01 10:00:00,1001,EU,books,25.0
2,2024-01-01 10:05:00,1002,US,electronics,199.0
3,2024-01-01 10:10:00,1003,EU,books,30.0
```

- **Parquet**:
  - columnar
  - compressed
  - efficient for aggregation
  - Example: daily revenue analysis, BI queries, data lake tables

```
transaction_id: [1, 2, 3]
timestamp: [t1, t2, t3]
user_id: [1001, 1002, 1003]
region: [EU, US, EU]
category: [books, electronics, books]
amount: [25.0, 199.0, 30.0]
```

- Key idea: Format affects performance

Partitioning by Column

- Large datasets are often stored in separate folders/prefixes
  - According to selected attribute values
- Example:
  - e.g., `region=EU/, region=US/, region=APAC/`
- Why?
  - Queries often filter by region or date
  - Spark can read only relevant parts
  - Less data scanned = faster analytics
- Trade-off: too many partitions may create too many small files
  - Spark can reduce output file count using `coalesce()`

Basic Examples

Typical workflow:

- Load data into DataFrame

```
df = spark.read.csv("people.csv", header=True)
df.show()
```

- Create temporary view

```
df.createOrReplaceTempView("people")
```

- Run SQL queries

```
r1 = spark.sql("SELECT name FROM people WHERE age > 18")
r2 = spark.sql("SELECT age, COUNT(*) FROM people GROUP BY age")
r3 = spark.sql("SELECT * FROM people ORDER BY age DESC")
```

- Work with results

```
r1.show()
r2.filter(r2.cnt > 10).show()
r3.show()
```

SQL and DataFrame API

- Two ways to work with structured data in Spark
- Both approaches produce the same execution plan
- When to use what?

SQL

- familiar relational queries
- joins, aggregations
- analysts & DB mindset

```
spark.sql(
"SELECT name, age "
"FROM people "
"WHERE age > 18"
).show()
```

DataFrame API

- programmatic pipelines
- integration with Python
- complex transformations

```
df.filter(df.age > 18) \
    .select("name", "age") \
    .show()
```

DataFrame API – Common Operations (PySpark)

- Selection & columns
  - `select()` – choose columns
  - `withColumn()` – create/modify column
  - `drop()` – remove column
- Filtering: `filter()` / `where()` – filter rows
- Aggregation
  - `groupBy()` – grouping
  - `agg()` – aggregate functions (count, avg, sum, …)
- Joins: `join()` – join DataFrames
- Sorting: `orderBy()` / `sort()` – sorting
- Output
  - `show()` – display results
  - `write()` – save data (CSV, Parquet, …)

https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/dataframe.html

Spark + Object Storage (MinIO)

Compute and storage are separated

- Spark does not require local storage.
  - Reads and writes data via storage connectors
- Architecture
  - **Compute layer:**
    - Apache Spark cluster
    - Executes distributed jobs
  - **Storage layer:**
    - MinIO (S3-compatible object storage)
    - Stores raw and processed data
- Connection:
  - `s3a:// protocol` (Hadoop S3 connector)

```
df = spark.read.csv("s3a://data/raw/users.csv",
     header=True
)
df.write.mode("overwrite").parquet("s3a://data/output/")
```

References

- Apache Hadoop: http://hadoop.apache.org/
- Hadoop: The Definitive Guide, by Tom White, 2nd edition, Oreilly's, 2010
- Hadoop Map/Reduce Tutorial
  - http://hadoop.apache.org/docs/r0.20.2/mapred_tutorial.html
- David DeWitt and Michael Stonebraker: Relational Database Experts Jump The MapReduce Shark
- Spark Overview
  - https://spark.apache.org/docs/latest/index.html
- Apache Spark Examples
  - https://spark.apache.org/examples.html
- A Tale of Three Apache Spark APIs: RDDs, DataFrames, and Datasets
  - https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html