Data Integration and Large Scale Analysis

Slides credit: Matthias Boehm - Shafaq Siddiqi

09- Distributed Data-Parallel Computation

Lucas Iacono. PhD. - 2024

Part B

Large-Scale Data Management & Analysis

- LU3. Cloud Computing
    - Cloud Computing Fundamentals [Nov 29]
    - Cloud Resource Management and Scheduling [Dec 06]
    - Distributed Data Storage[Dec 13]

Part B

Large-Scale Data
Management & Analysis

- LU4. Large-Scale Data Analysis
  - **Distributed, Data-Parallel Computation [Dec 20]**
  - Distributed Stream Processing **[Jan 10]**
  - Distributed Machine Learning Systems **[Jan 17]**

Agenda

- Announcements
- Data-Parallel Collection & Processing
- Data-Parallel DataFrame Operations

Announcements

Announcements

Course Evaluation and Exam

- Course evaluation: 20/02/2025
- Exam date: Feb 07, 3:00pm (90 min written exam)
- Oral Exam for Erasmus Students
    - Schedule available in TeachCenter **(23/12/2024)**

Motivation and Terminology

Motivation and Terminology

Recap: Distributed Collections

Logical multi-set (bag) of key-value pairs (unsorted collection)

Different physical representations key-value pairs can be stored in various ways (e.g., database, across files, or in memory).

Easy Distribution via Horizontal Partitioning. Data divided into "chunks" (shards or partitions) based on the keys. Each chunk stored on a different machine (easier to handle large-scale data).

How collections are created: from single file with data or a folder of files (even if they're messy and unsorted).

Key	Value
13:00:01	12.1
14:00:05	16.0
13:00:03	12.5
13:00:05	13.0
14:00:04	15.7
14:00:06	16.3
13:00:00	12.1

Motivation and Terminology

Recap: Files and Objects

File: large and continuous block of data saved in a specific format (CSV, Binary, etc.).

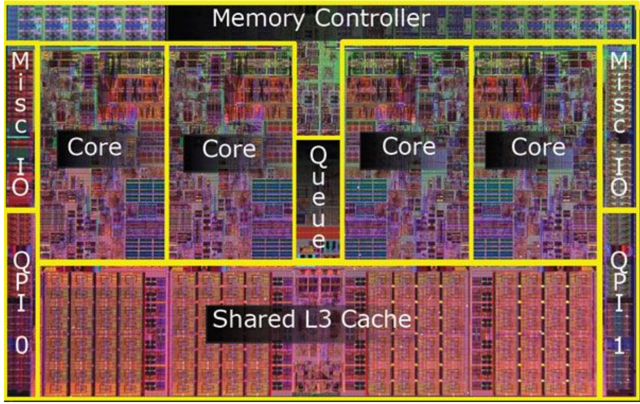
Object: like a file, but binary and it comes with metadata (Images on S3)

Motivation and Terminology

Recap: Object Storage

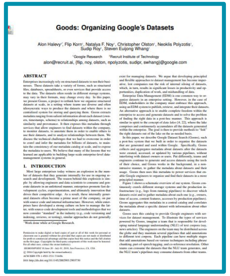
1. **Object Storage (e.g. AWS S3):**
    a. Data stored as objects (data, metadata, and UID).
    b. Ideal for storing unstructured data like media files, backups, or large datasets.
    c. Objects of a limited size (e.g., 5TB in AWS S3).

Motivation and Terminology



Nehalem Architecture

- **Integrated Memory Controller:** Integrated in chip, -- latency and ++ memory performance.
- **Support for DDR3 Memory:** Higher memory bandwidth (compared to DDR2).
- **QuickPath Interconnect (QPI):** High-speed, point-to-point connection (no Front-Side Bus).
- **Enhanced Hyper-Threading:** Each core supports two threads (+++ performance)
- **Multi-Core Scalability:** 2 to 8 cores per processor (2 threads / core)
- **Improved Cache Design:** Dedicated L1 and L2 cache p/core shared L3 cache


Goods: Organizing Google's Datasets

Alon Halevy[2], Flip Korn[1], Natalya F. Noy[1], Christopher Olston[1], Neoklis Polyzotis[1], Sudip Roy[1], Steven Euijong Whang[1]

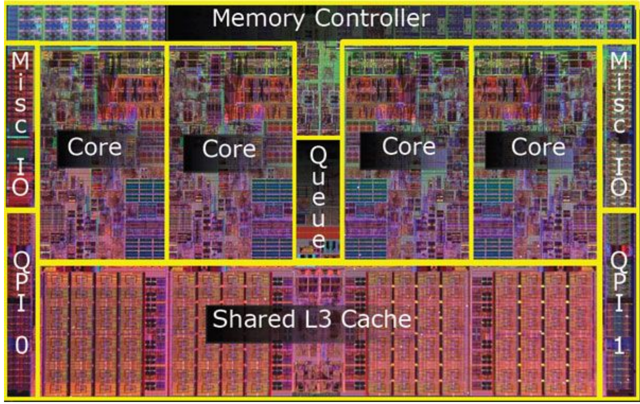
[1]Google Research [2]Recruit Institute of Technology

alon@recruit.ai, {flip, noy, olston, npolyzotis, sudipr, swhang}@google.com

ABSTRACT

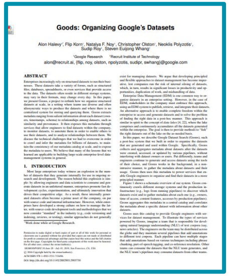
Michael E. Thomadakis:
The Architecture of the
Nehalem Processor and NehalemEP SMP Platforms, Report, 2010

Motivation and Terminology



Nehalem Architecture

- **Energy Efficiency:** Turbo Boost for dynamic clock speed adjustments.
- **Advanced Manufacturing Process:** Higher transistor density and better efficiency.
- **Integrated Graphics (in later models):** Some models included integrated GPUs.
- **Foundation for Modern Architectures:** Established the groundwork for subsequent Intel architectures like Sandy Bridge and Skylake.


Goods: Organizing Google's Datasets

Alon Halevy[2], Flip Korn[1], Natalya F. Noy[1], Christopher Olston[1], Neoklis Polyzotis[1], Sudip Roy[1], Steven Euijong Whang[1]

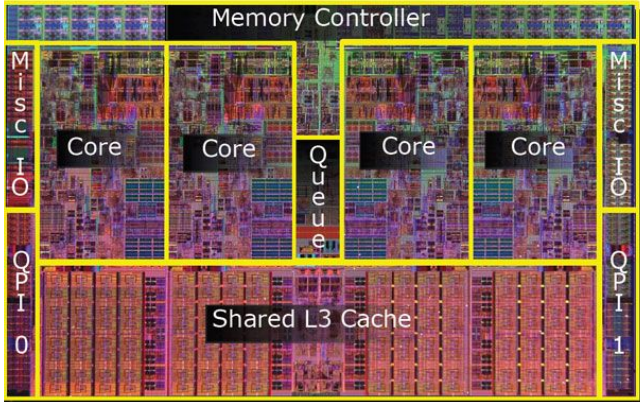
[1]Google Research [2]Recruit Institute of Technology

alon@recruit.ai, {flip, noy, olston, npolyzotis, sudipr, swhang}@google.com

ABSTRACT

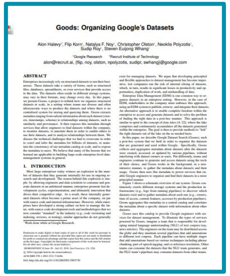
Michael E. Thomadakis: The Architecture of the Nehalem Processor and NehalemEP SMP Platforms, Report, 2010

Motivation and Terminology



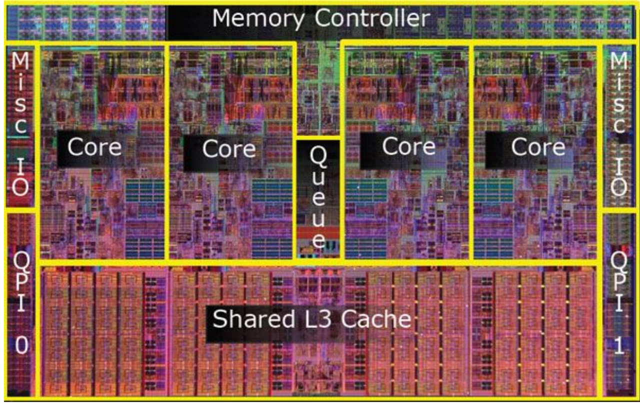
Nehalem Architecture

- **Pipeline**
  - Frontend:
    - Instruction Fetch
    - Pre-Decode
    - Decode CISC 2 uOps **(ADD [eax], 5)**
      - Load the value from memory.
      - Add 5 to the loaded value.
      - Store the result back to memory.
  - Backend:
    - Rename/Allocate
    - Scheduler
    - Execute


Goods: Organizing Google's Datasets

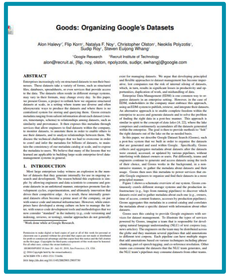
Michael E. Thomadakis: The Architecture of the Nehalem Processor and NehalemEP SMP Platforms, Report, 2010

Motivation and Terminology



Nehalem Architecture

- **Out-of-Order**
  - Instructions are not necessarily executed in the order they appear in the program
- **Execution Engine:** 4 Inst x Cycle (IPC=4)
- **128-bits Floating-point multiplication**
- **128-bits floating-point addition**



Michael E. Thomadakis: The Architecture of the Nehalem Processor and NehalemEP SMP Platforms, Report, 2010

Motivation and Terminology

	Single Data	Multiple Data
Single Instruction	SISD (uni-core)	SIMD (vector)
Multiple Instruction	MISD (pipeline)	MIMD (multi-core)

Flynn's Classification

Computer architectures based on how they handle instructions and data.

- **SISD:**
  - One task at time - one data chunk (e.g. PC running a single program)
- **SIMD:**
  - One task at time - multiple data chunks (e.g. GPUs rendering)
- **MISD:**
  - Multiple tasks - one data chunk (e.g. fault-tolerant computers)
- **MIMD:**
  - Multiple tasks - multiple data chunks (multi-core CPUs 1 Core -> Program)

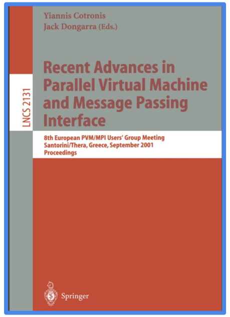
Michael J. Flynn, Kevin W. Rudd: Parallel Architectures.
ACM Comput. Surv. 28(1) 1996

Motivation and Terminology

Distributed, Data-Parallel Computation

- Parallel computation of function **foo()** ➔ **single instruction**
    - A **single function** applied to **all data items** in parallel.
- Collection X of data items (**key**-**value** pairs) ➔ **multiple data**
    - foo() operates on multiple pieces of data (**key**-**value** pairs).
- **Data parallelism** similar to **SIMD** but more coarse-grained notion of “instruction” and “data” ➔ **SPMD** (single program, multiple data)

Y = **X.map**(**X** **->** **foo(x)**) **X** = Data Items (e.g. array), **.map** (operation to each element in **X**), **Y** = Output

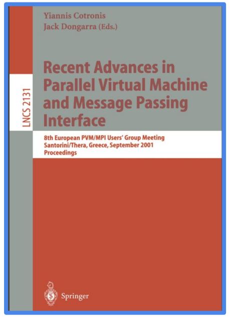


[Frederica Darema: The SPMD Model : Past, Present and Future. PVM/MPI 2001]

Motivation and Terminology

SPMD

- **Dynamic Work Assignment.** Processes can **self-schedule**, ++ **flexibility & efficiency**.
- **More General than SIMD.** SPMD allows **different instruction streams for different data**. It can handle more complex tasks.
- **Efficient Control.** Performed at the **application level** rather than the OS level (less costly and more efficient than **F&J**.
- **Applications:**
  - **MPI (Message Passing Interface)**
  - **PVM (Parallel Virtual Machine)**
  - **Grid Computing**



[Frederica Darema: The SPMD Model : Past, Present and Future. PVM/MPI 2001]

Motivation and Terminology

Model	Key Features	Pros	Cons
BSP (Bulk Synchronous)	Global barriers enable synchronization after each phase	+++ Correctness and consistency; simple to implement	Overhead due to waiting at barriers Slow for stragglers
ASP (Asynchronous Parallel)	Processes run independently	Faster execution (no waiting)	Accuracy issues from outdated data
SSP (Stale-Synchronous Parallel)	Controlled staleness allows fastest processes to proceed within a limit	Balances efficiency and consistency; reduces waiting time compared to BSP	Small inaccuracies

Data-Parallel Collection & Processing

Hadoop

Brief Hadoop History

- Google's GFS + MapReduce [ODSI'04] -> Apache Hadoop (2006).
- Apache Hive (SQL), Pig (ETL), Mahout (ML), Giraph (Graph)

Hadoop Architecture / Eco System

- Management (Ambari)
- Coordination / workflows (Zookeeper, Oozie)
- Storage (HDFS)
- Resources (YARN)

Hadoop

Hadoop Ecosytem

- Apache Hive (SQL)
  - **What it is:**
    - Data warehouse infrastructure built on top of Hadoop.
    - Allows you to query and analyze large datasets stored in Hadoop using a SQL-like language called HiveQL.
  - **Main Purpose:**
    - Querying and analysis of big data using familiar SQL syntax.
    - Suitable for batch processing and data summarization.
  - **Use Case:**
    - Running SQL queries to analyze log data or generate business reports.

Hadoop

Hadoop Ecosytem

- Apache Pig (ETL)
    - **What it is:**
        - **High-level** platform for creating **data processing programs** in Hadoop.
        - **Pig language** to **simplifies the MapReduce** jobs writing process.
    - **Main Purpose:**
        - ETL operations. Cleaning, transforming, and preparing large datasets for analysis.
    - **Use Case:**
        - Processing raw web logs into structured formats for further analysis.

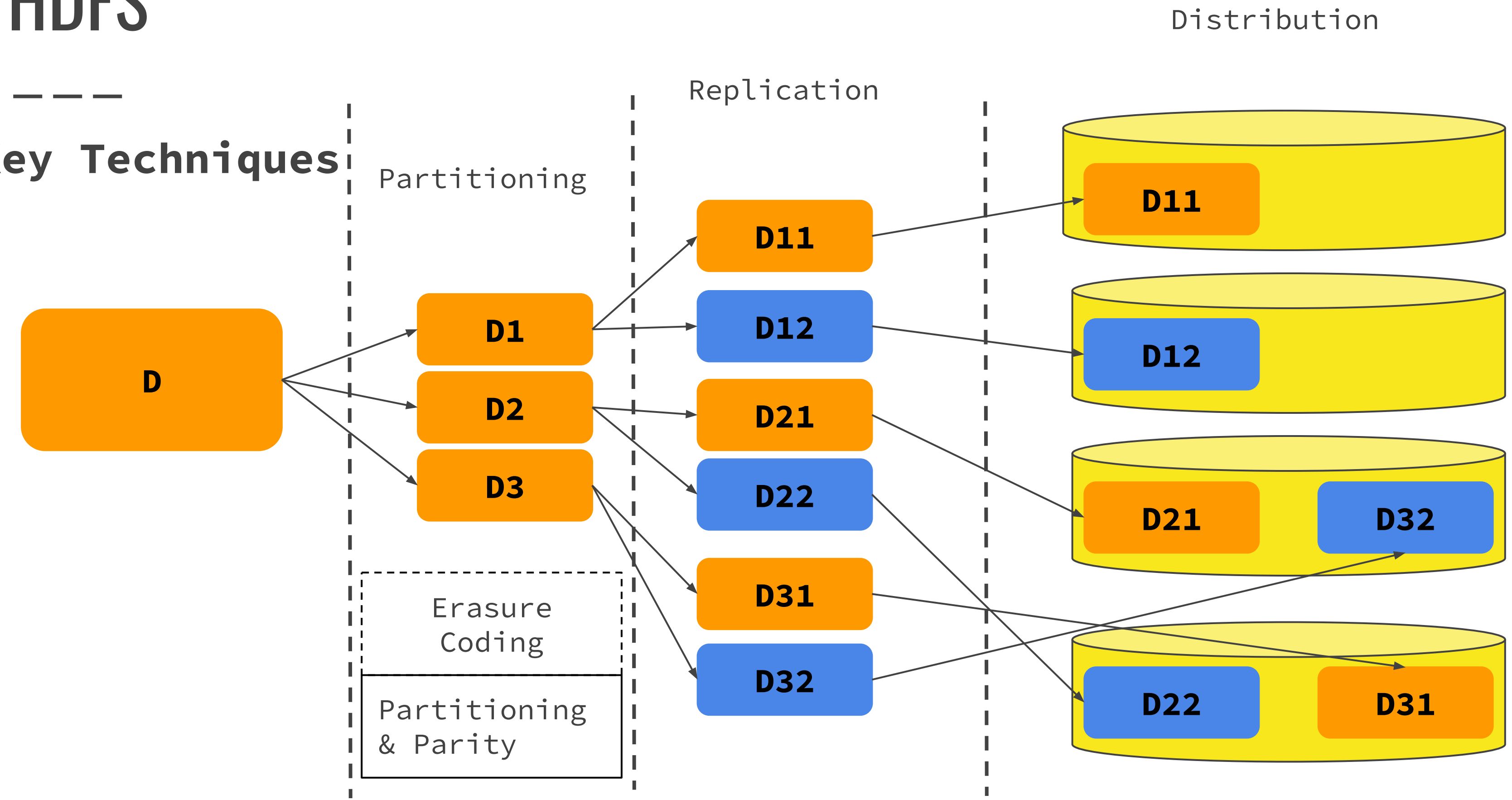
Hadoop

Hadoop Ecosytem

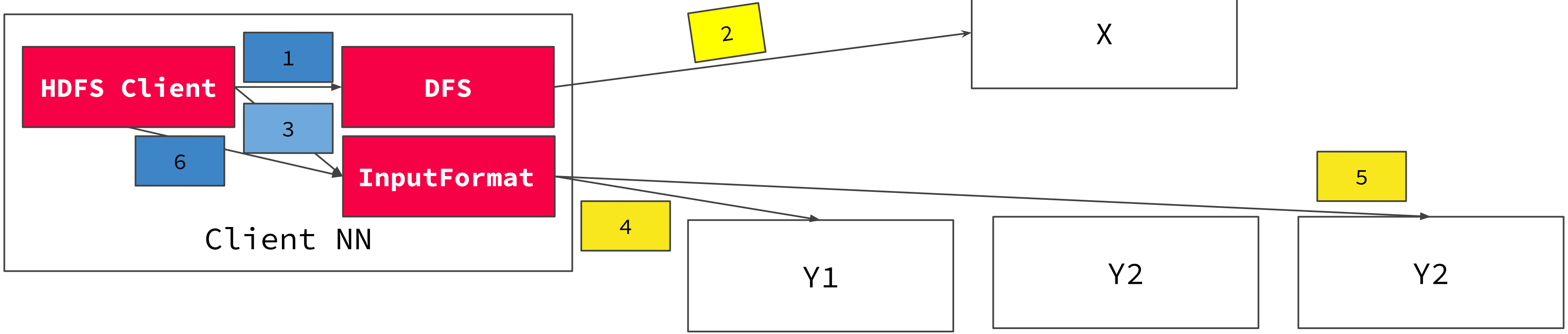
- Apache Mahout (ML)
  - **What it is:**
    - A library for building scalable **machine learning** algorithms on top of Hadoop.
    - Focused on distributed or scalable implementations of common ML algorithms.
  - **Main Purpose:**
    - Implementing machine learning algorithms like clustering, classification, and recommendation systems on large datasets.
  - **Use Case:**
    - Building a recommendation system for an e-commerce platform using collaborative filtering.

HDFS

Key Techniques



Recap: HDFS Read



1. Open
2. Get Block Locations
3. Read
4. Read
5. Read
6. Close

MapReduce – Programming Mode

Overview

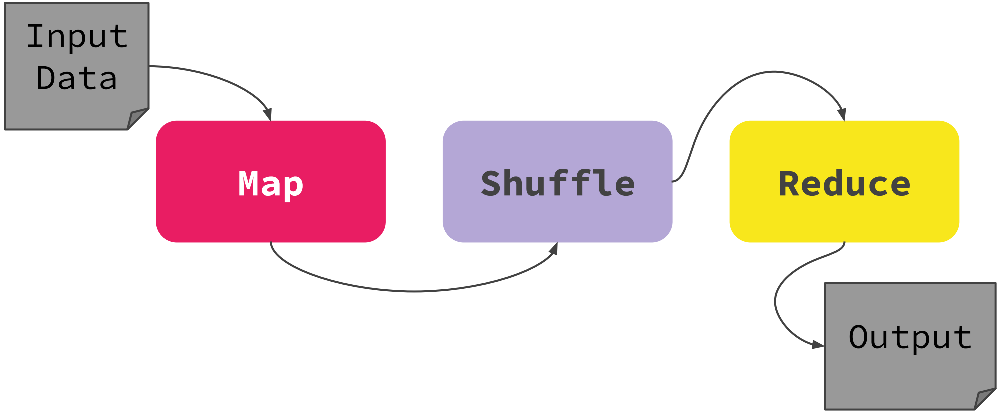
- MapReduce is a programming model for processing large datasets in parallel, distributed across multiple nodes.
- Developed by Google; popularized by Apache Hadoop.

Map

Processes and transforms input data into intermediate key-value pairs.

Reduce

Aggregates intermediate data to produce the final result.



MapReduce I

Why MapReduce?

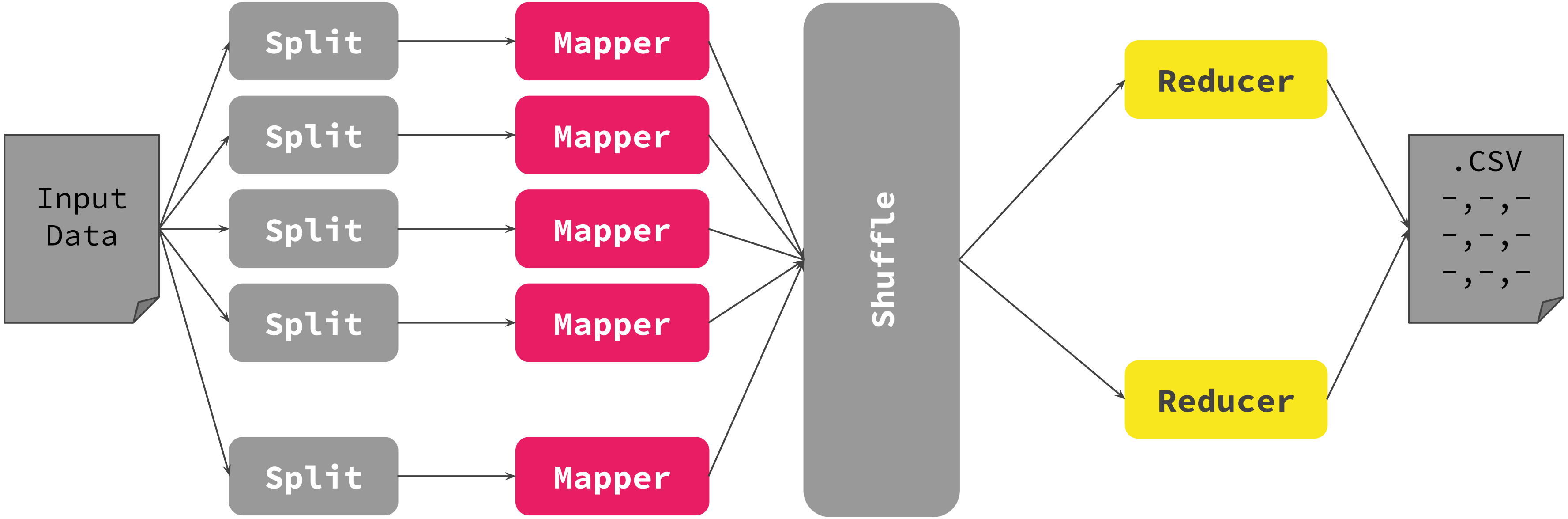
- Handles large-scale data processing efficiently.
- Works on commodity hardware.
- Built-in fault tolerance.
- Suitable for structured, semi-structured, and unstructured data.

MapReduce II

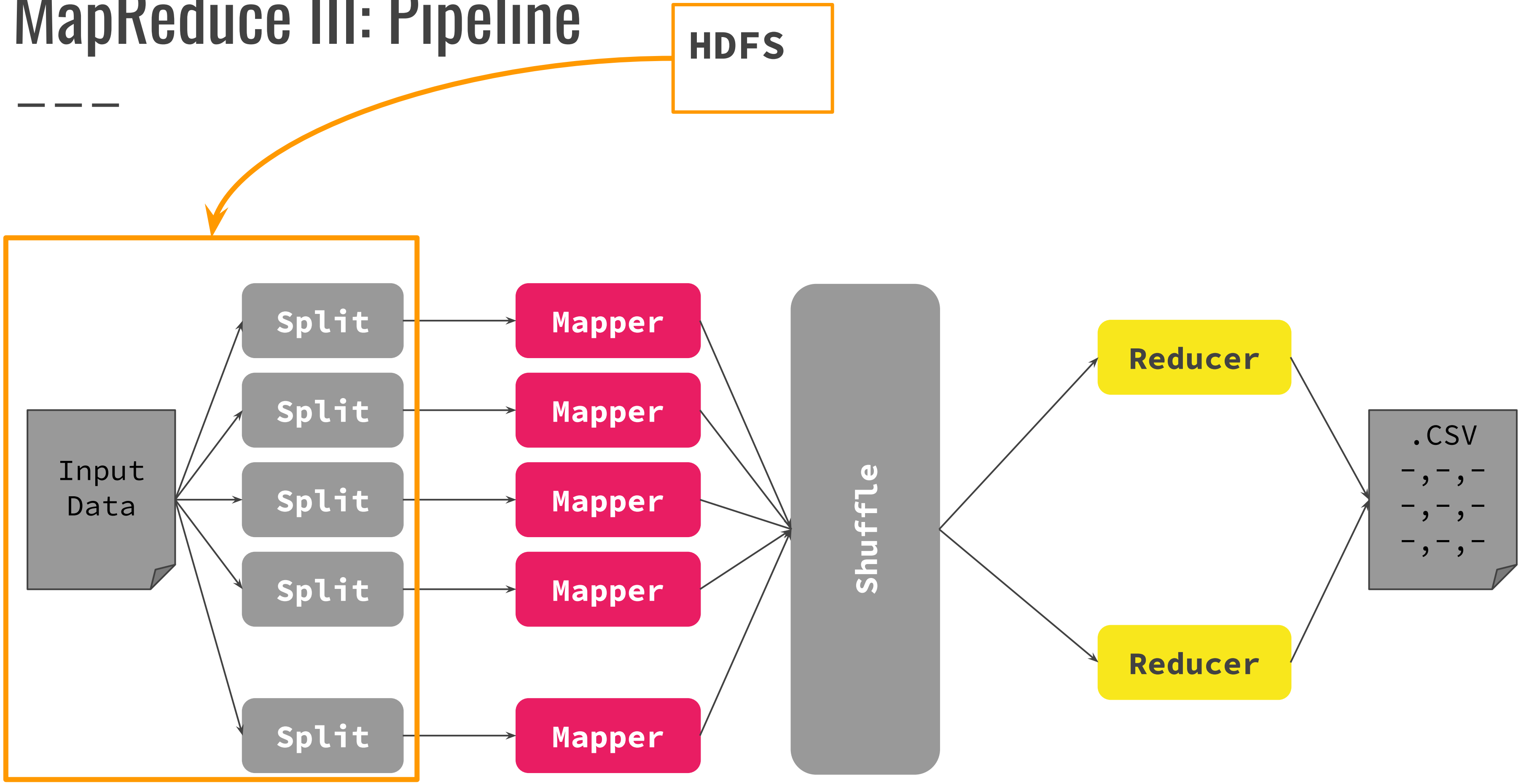
Key Concepts

- Distributed Processing: Data is split across multiple nodes for parallel execution.
- Key-Value Pairs: Core data structure in MapReduce.

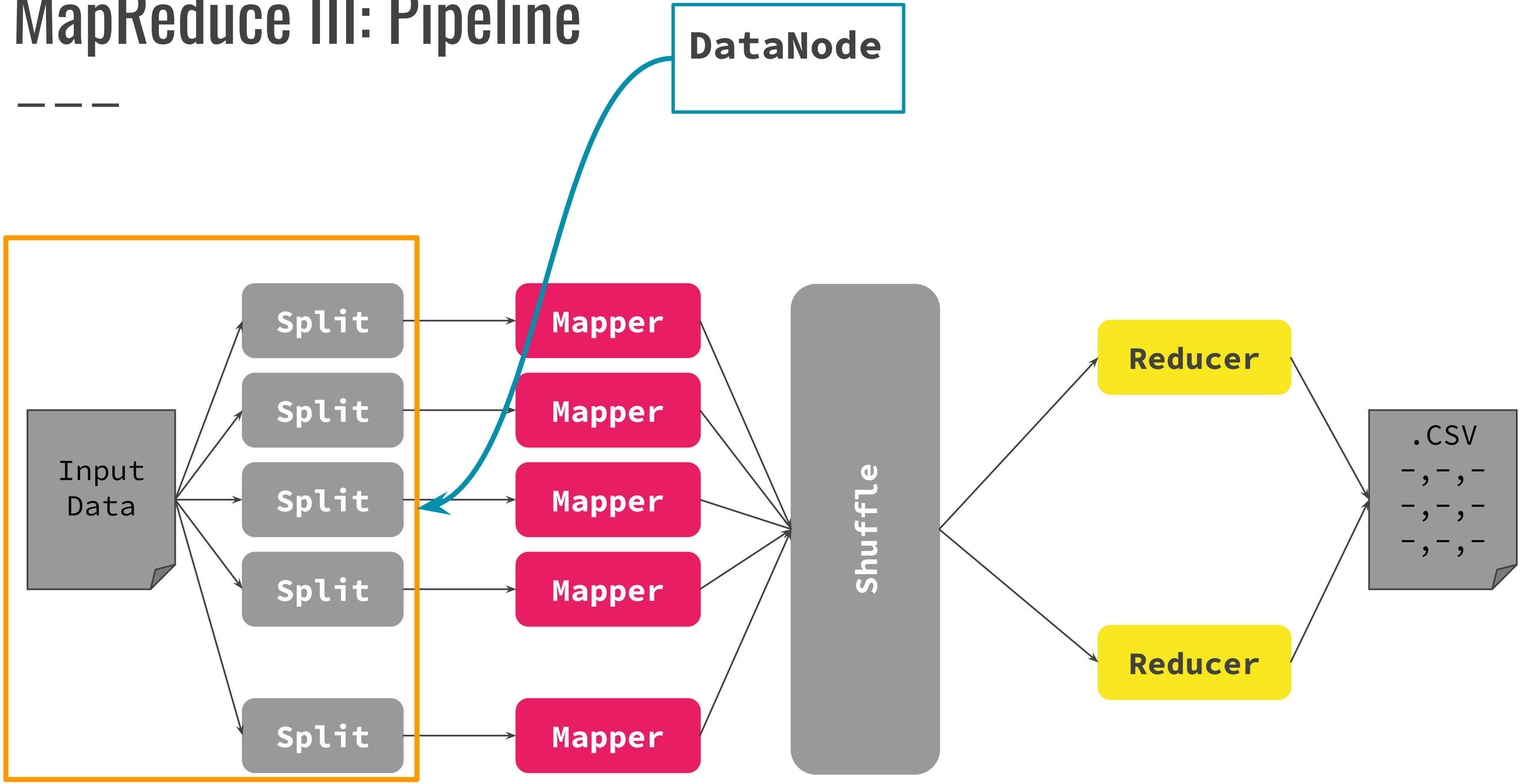
MapReduce III: Pipeline



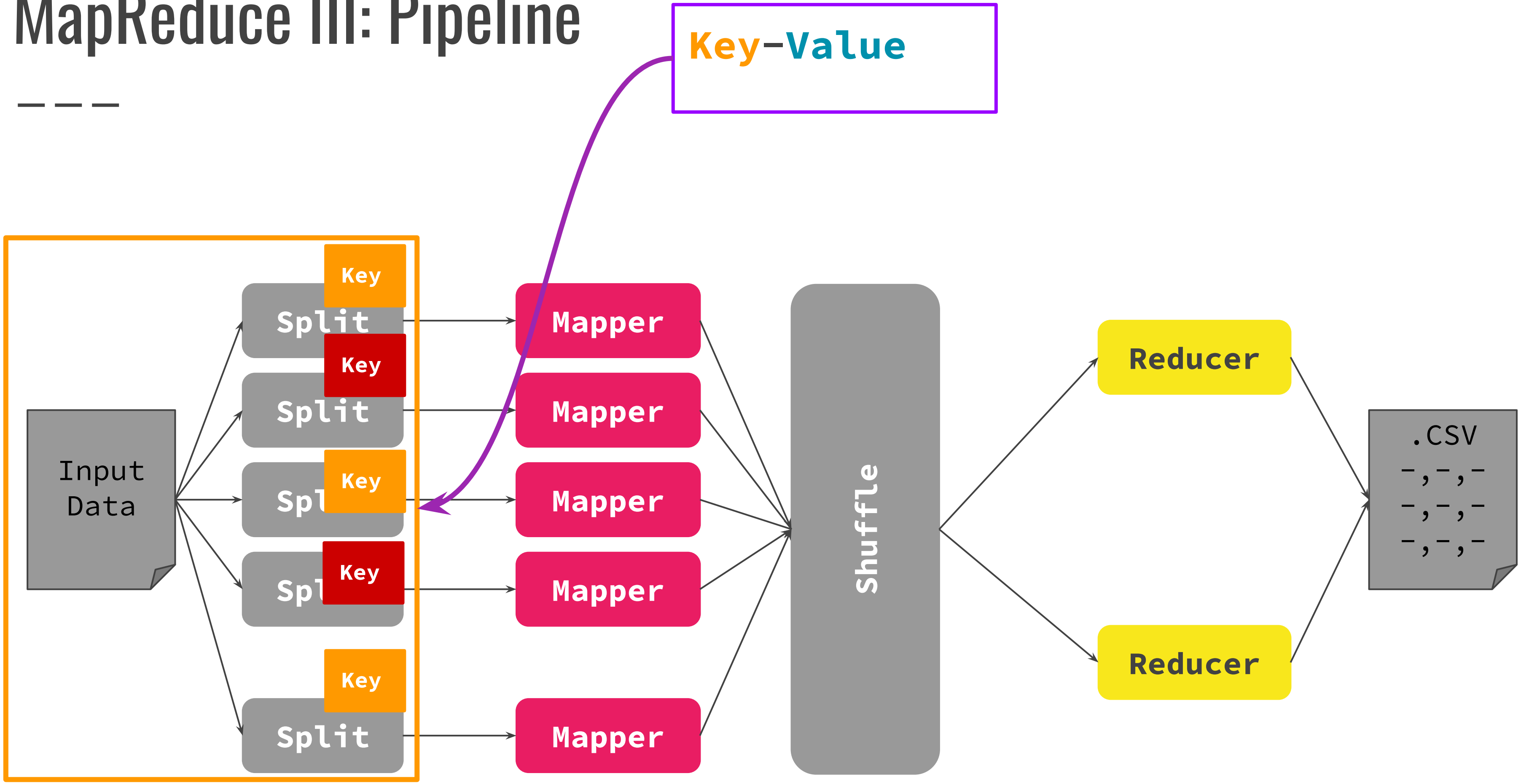
MapReduce III: Pipeline



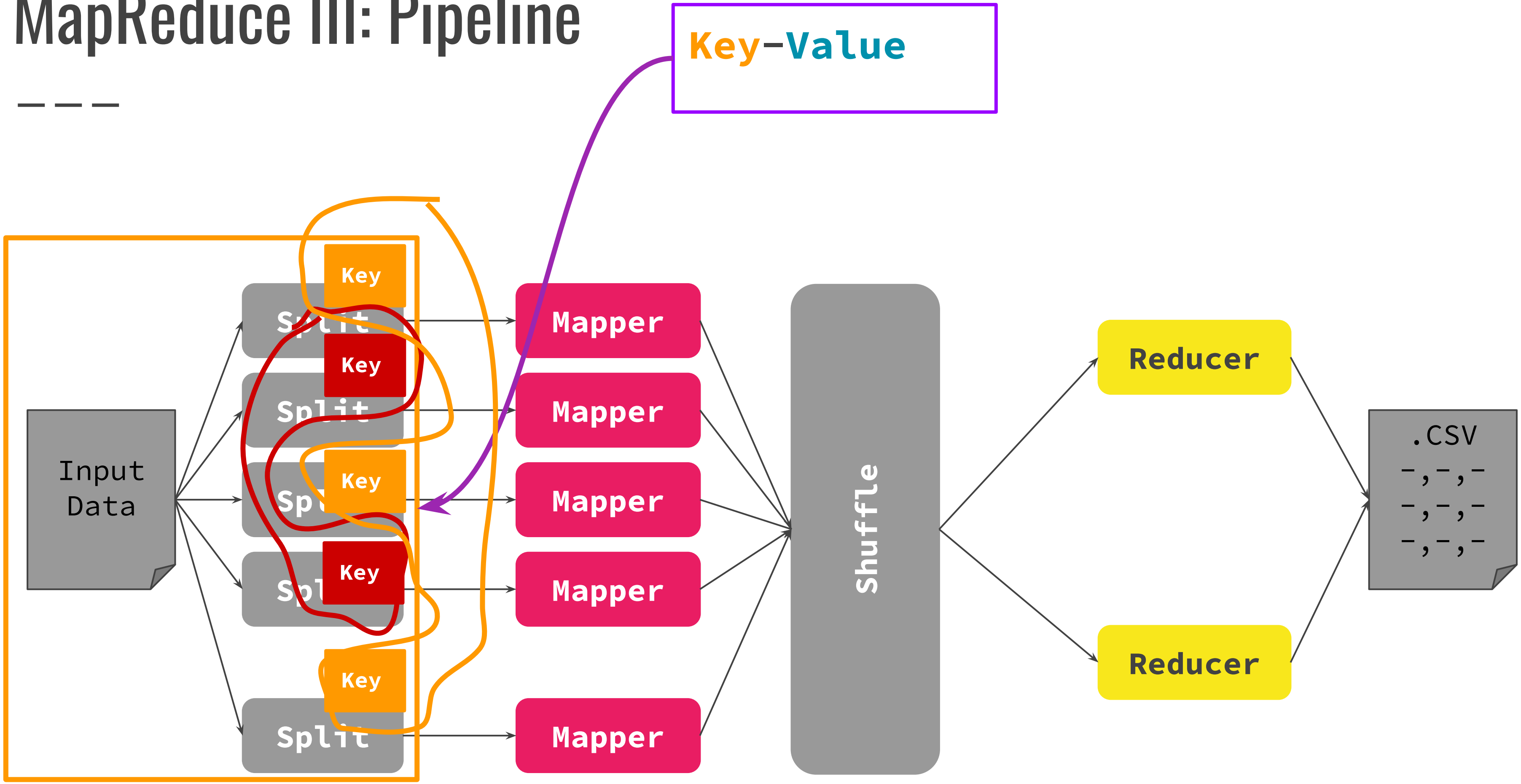
MapReduce III: Pipeline



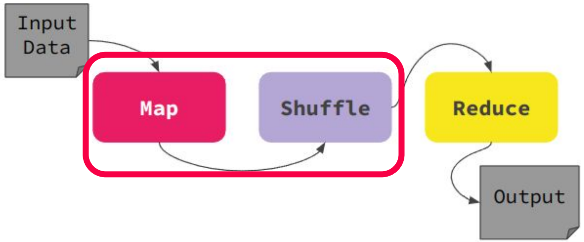
MapReduce III: Pipeline

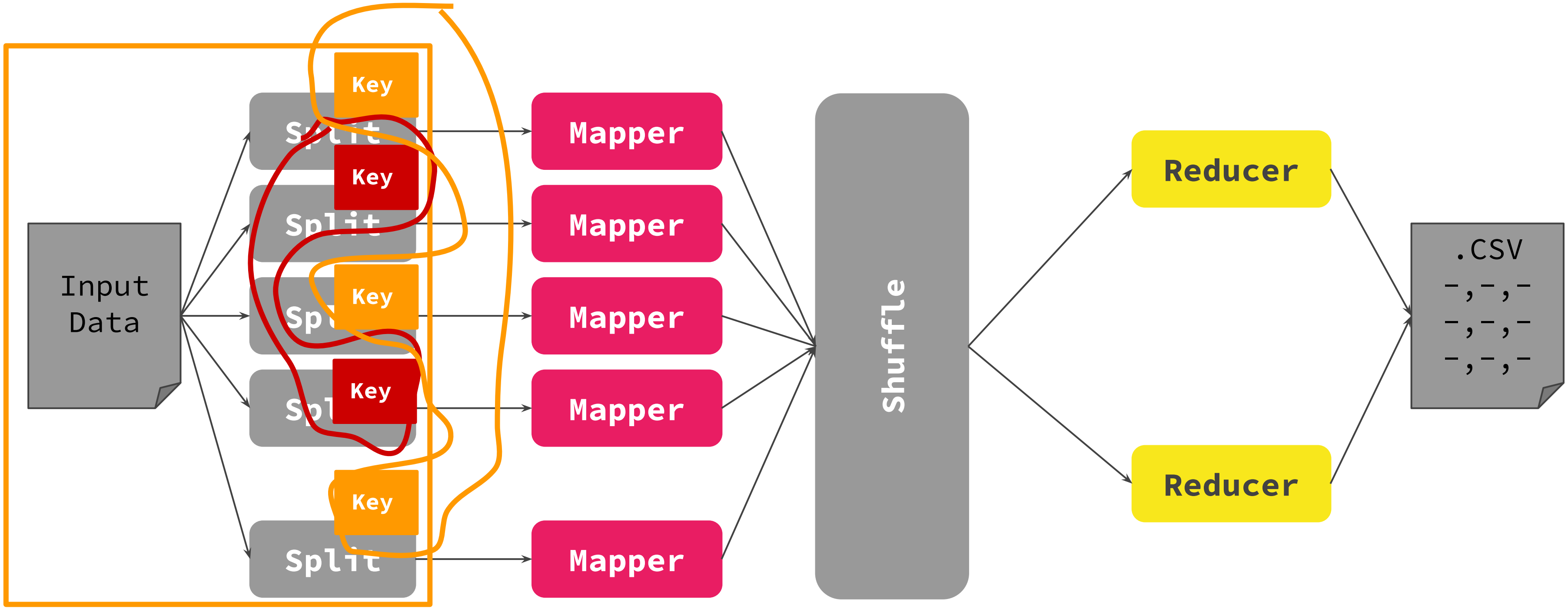


MapReduce III: Pipeline

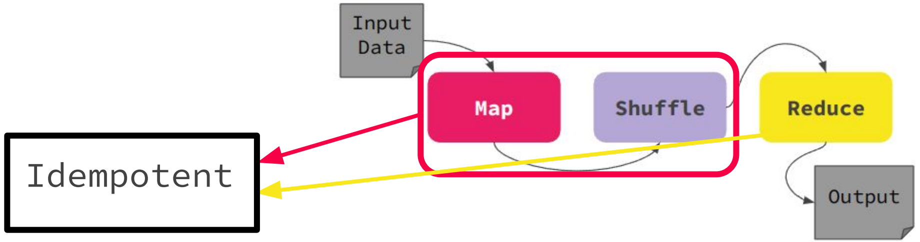


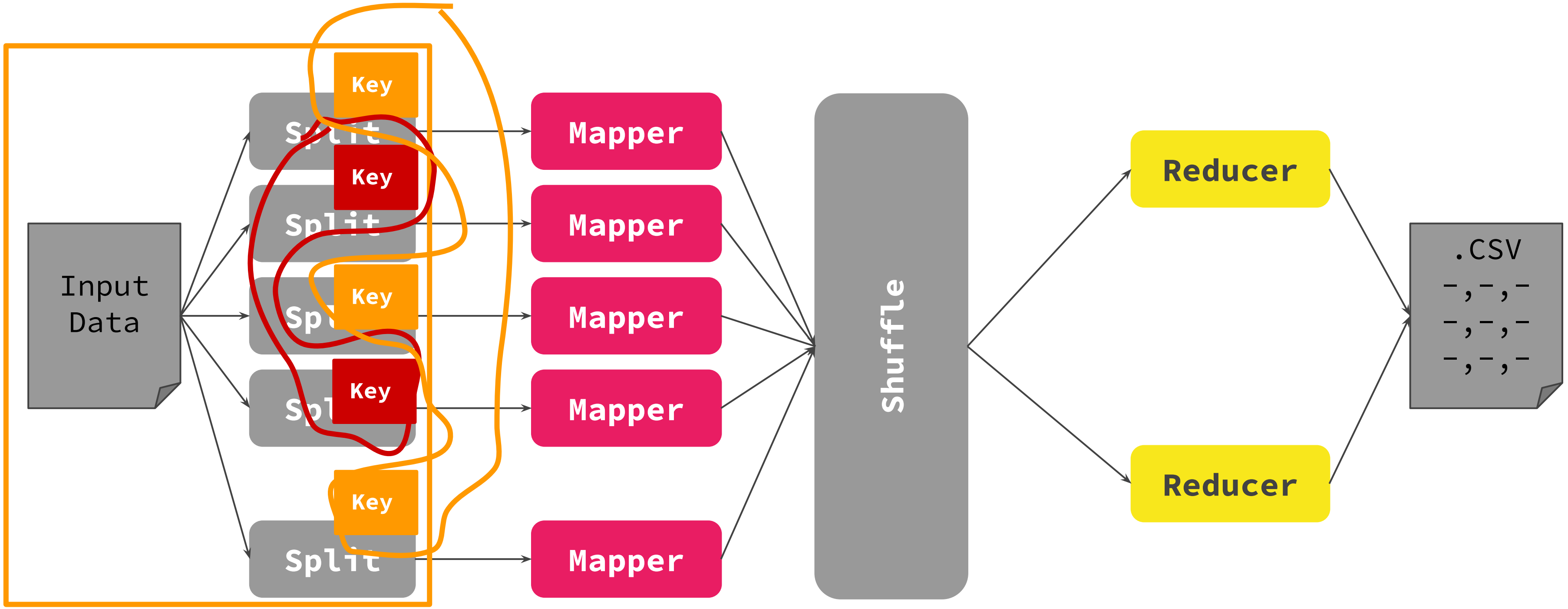
MapReduce III: Pipeline



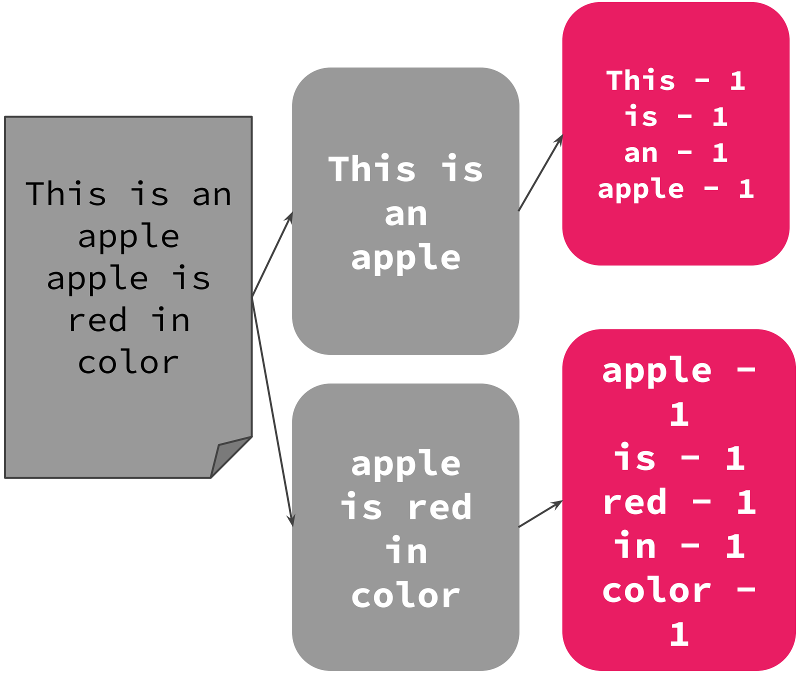


MapReduce III: Pipeline



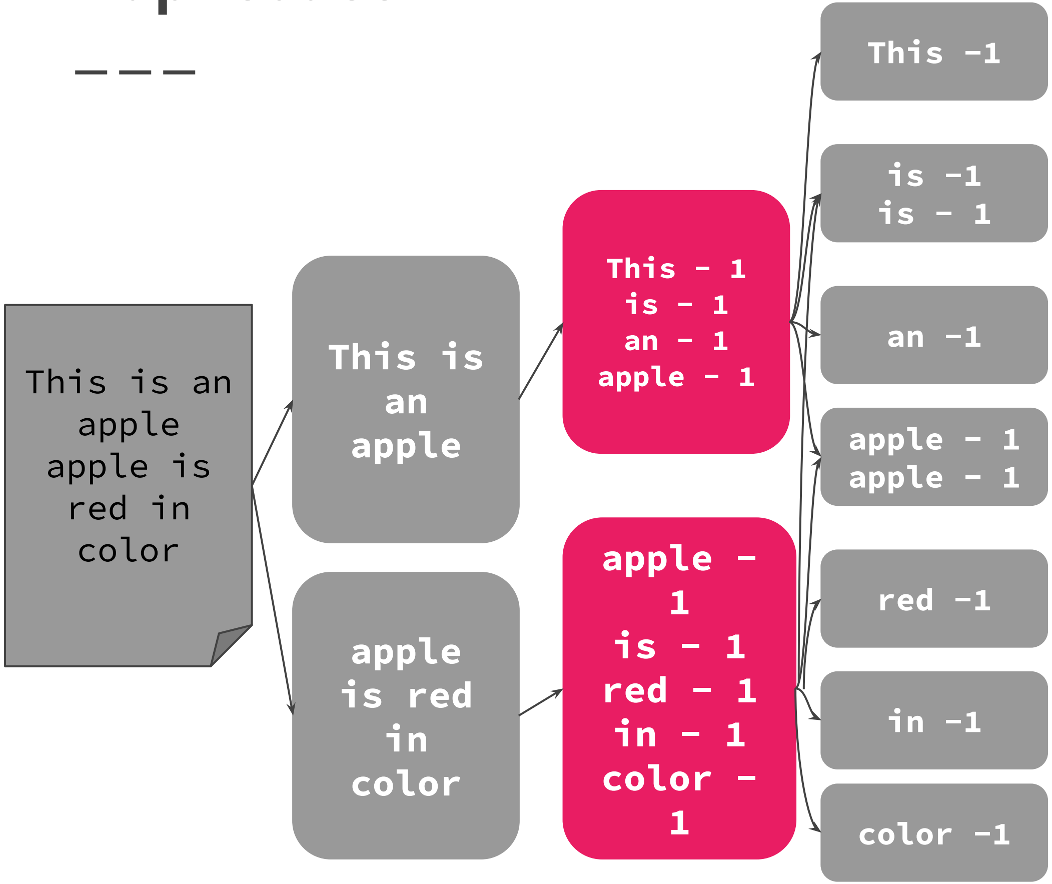


MapReduce IV

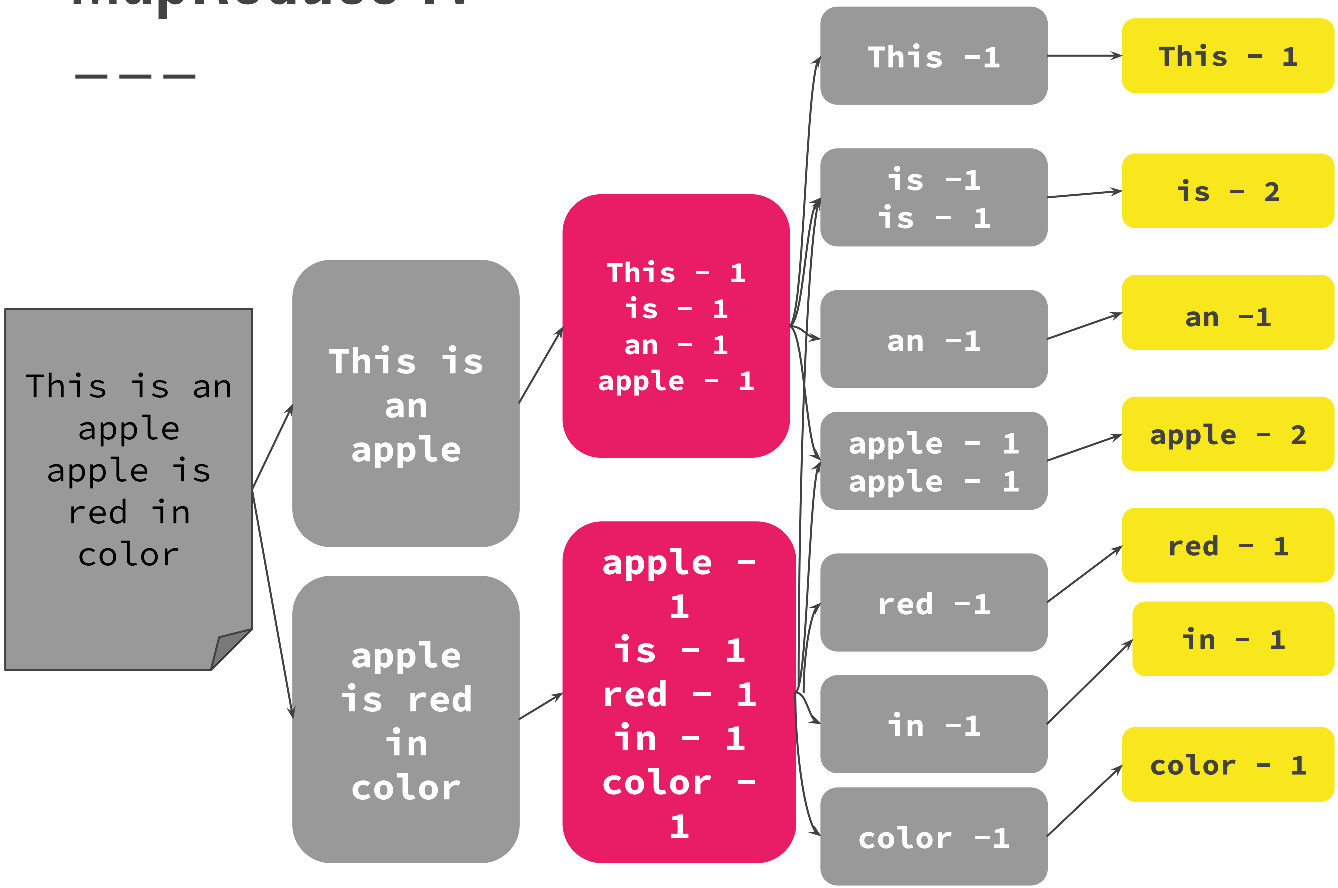


[Jeffrey Dean, Sanjay Ghemawat: MapReduce: Simplified Data Processing on Large Clusters. OSDI 2004]
]

MapReduce IV

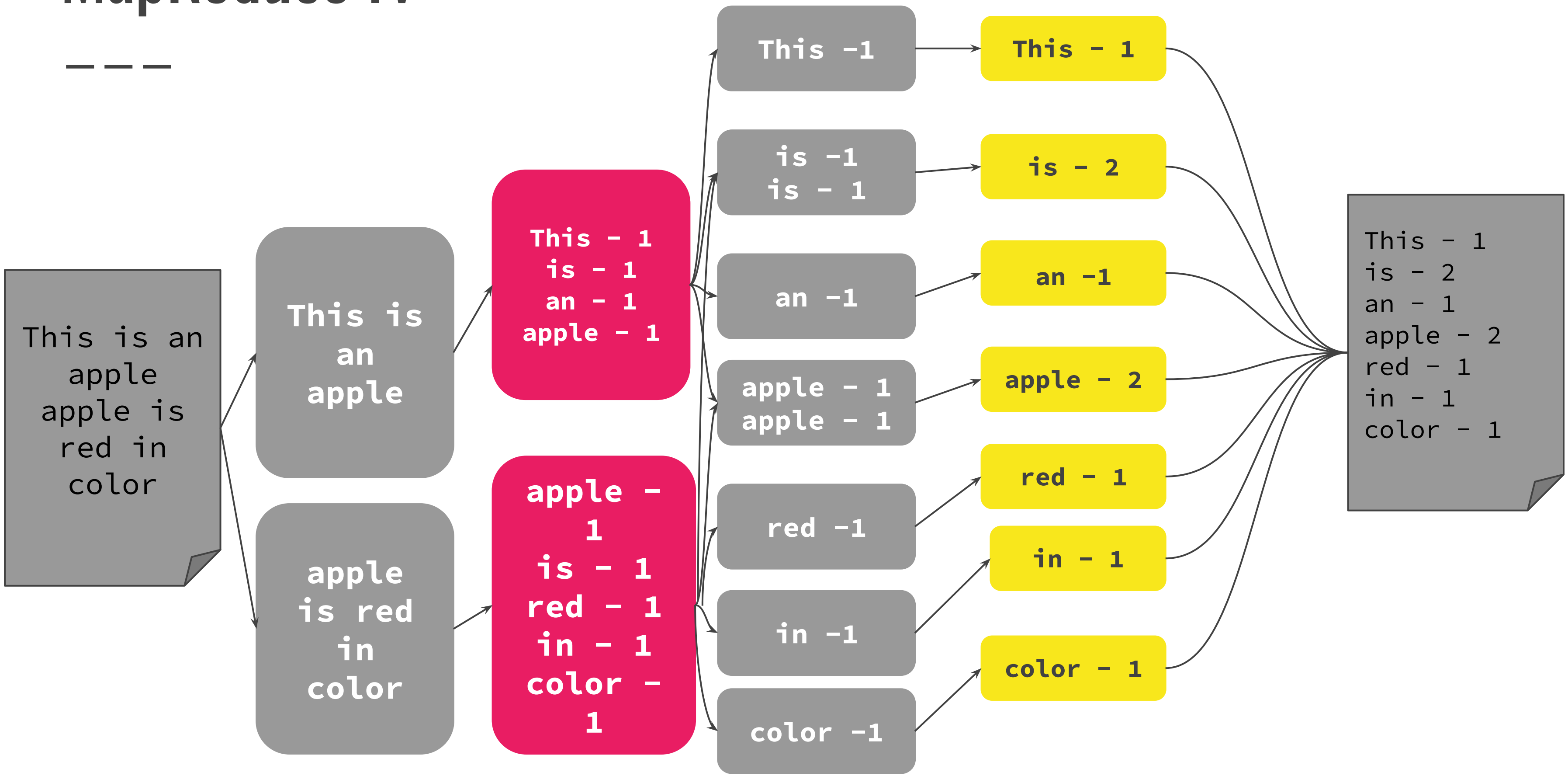


MapReduce IV



This - 1

MapReduce IV



MapReduce VI: Hands on Lab

Servers Log

- Use the **MapReduce** programming model to:
    - Count how many times each page was accessed.
    - Identify the most popular page.
- Calculate the Average Using **MapReduce**
    - Given a list [4, 8, 15, 16, 23, 42], compute the average using MapReduce.

MapReduce: Summary (Pros)

- **Large-scale processing.** Large amounts of data distributed across multiple nodes in a cluster.
- **Fault-tolerant.** If a node fails, the system can recover and reassign tasks to other nodes.
- **User Defined Functions and files.** Developers can define their own **custom processing logic through UDFs**, and the **model relies on files** to store intermediate and final results.
- **Flexibility.** Developers can **customize processing logic** while the system manages **distribution** and **fault recovery automatically**.
- **Restricted functional APIs.** MapReduce relies on a limited set of functional primitives:
  - **Map**: Transforms input data into key-value pairs.
  - **Reduce:** Aggregates values associated with the same keys to produce results.
- **Implicit parallelism**. Developers only need to implement the Map and Reduce functions; the distribution of workload across nodes relies on the system.

MapReduce: Summary (Cons)

- **Performance:** its performance can suffer in complex workloads due to heavy reliance on I/O (writing and reading intermediate data to/from disk).
- **Low-level APIs:** The API is relatively basic, requiring a lot of manual effort to implement more sophisticated workflows.
- **Many different systems:** Specialized systems (**e.g., Apache Spark**, Apache Flink, or distributed database systems) have emerged as alternatives, often being more efficient and user-friendly.

Spark History and Architecture

Evolution to Spark (and Flink)

- Spark [HotCloud'10] + Resilent Distributed Data Sets (RDDs) [NSDI'12] → Apache Spark (2014)
- **Design 1: Standing executors with in-memory storage:**
  - Spark keeps **long-running worker processes** (executors) active, enabling tasks to run faster by avoiding repeated setup costs.
  - Data is stored in memory whenever possible, **minimizing disk I/O** for iterative and interactive jobs.
- **Design 2: Lazy evaluation:**
  - **Directed Acyclic Graph of transformations** rather than executing them immediately.
  - Actions (e.g., collect, save, count) trigger DAG's execution, allowing workflow optimization by reordering and combining operations.

Spark History and Architecture

- **Design 3:** Fault tolerance via RDD lineage
  - Data partition lost -> Spark can recompute using **lineage** graph of transformations applied to the data (reliability without heavy replication).
- **Performance:**
  - **In-memory storage.** By keeping **intermediate data in memory**, Spark significantly reduces disk I/O (faster for iterative tasks e.g machine learning).
  - **Fast job scheduling.** Spark's scheduler operates with **low overhead**, enabling tasks to be scheduled in milliseconds (~100ms), compared to Hadoop's ~10 seconds per job.

Spark History and Architecture

- **APIs:**
    - **Richer functional APIs.** Wide range of functional operators (e.g., map, reduce, filter, groupByKey, flatMap) compared to Hadoop -> easier to write complex workflows.
    - **General computation DAGs.** Unlike MapReduce, which forces jobs into two rigid phases (map and reduce), Spark supports general DAGs for more flexible computation flows.
    - **High-level APIs (DataFrame/Dataset).** DataFrames and Datasets offer high-level abstractions that simplify working with structured data and enable query optimization.

Spark History and Architecture

- **Unified Platform.** Multiple workloads into a single platform:
  - Batch processing (similar to MapReduce)
  - Streaming (real-time data)
  - Machine learning (MLlib)
  - Graph processing (GraphX)
  - SQL queries (Spark SQL)

Spark Functionality: Core components

Resilient Distributed Datasets (RDDs):

- Distributed collections of objects (foundation for fault tolerance and parallelism.)

DataFrames and Datasets:

- Higher-level abstractions for structured and semi-structured data (Optimized via Spark's Catalyst engine).

Spark SQL:

- Query structured data using SQL.

MLlib:

- Machine learning library for scalable algorithms.

GraphX:

- Graph processing library.

Spark Functionality: Architecture

Driver Program:

- Defines the **application** and **coordinates tasks**.

Cluster Manager:

- Allocates resources (YARN, Mesos, Kubernetes).

Executors:

- Workers that execute tasks and store data partitions.

DAGs:

- Spark builds a logical execution plan before running tasks.

Spark Functionality: Workflow

- **Create RDD/DataFrame:** Load data into Spark from HDFS, S3, or other sources.
- **Transformations:** Apply operations (e.g., map, filter, groupBy).
- **Actions:** Trigger execution (e.g., collect, save).
- **Execution:** (a) Splits tasks across nodes, (b) Uses DAG to optimize execution.

Spark: Hands on Lab

Servers Log

- Use the **COLAB** to simulate Spark basic operations
- Let’s take a look into **Databricks…**

Data-Parallel DataFrame Operations

Origins of DataFrames

Recap: Data Preparation Problem

- **80% Argument:** 80-90% time for finding, integrating, cleaning data
- Data scientists prefer scripting languages and in-memory libraries

Python DataFrames:

- Python pandas DataFrame for seamless data manipulations (most popular packages/features)
- DataFrame: table with a schema
- Descriptive stats and basic math, reorganization, joins, grouping, windowing
- Limitation: Only in-memory, single-node operations

```
import pandas as pd

df = pd.read_csv('data/tmp1.csv',
index_col=2)

df.head()

df = pd.concat(df, df[['A','C']], axis=0)
```

Spark DataFrames and DataSets

Overview Spark DataFrame

- DataFrame is distributed collection of rows with named/typed columns
- Relational operations (e.g., projection, selection, joins, grouping, aggregation)
- DataSources (e.g., json, jdbc, parquet, hdfs, s3, avro, hbase, csv, cassandra)

DataFrame and Dataset APIs

- DataFrame was introduced as basis for Spark SQL
- DataSets allow more customization and compile-time analysis errors (Spark 2)

DataFrame and Dataset APIs

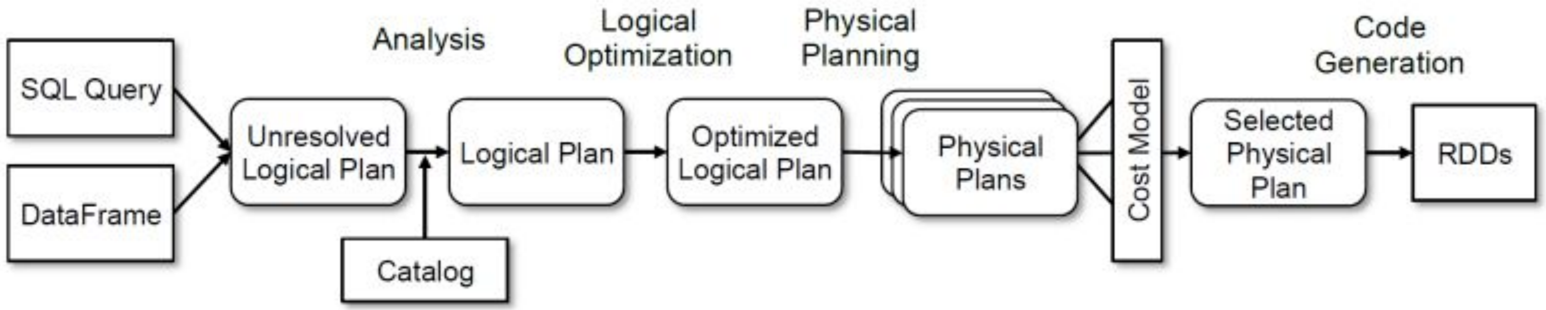
```
logs = spark.read.format("json").open("s3://logs")
logs.groupBy(logs.user_id).agg(sum(logs.time))
.write.format("jdbc").save("jdbc:mysql//...")
```

SparkSQL and DataFrame/Dataset

Overview SparkSQL

- Shark (~2013): academic prototype for SQL on Spark
- SparkSQL (~2015): reimplementation from scratch
- Common IR and compilation of SQL and DataFrame operations

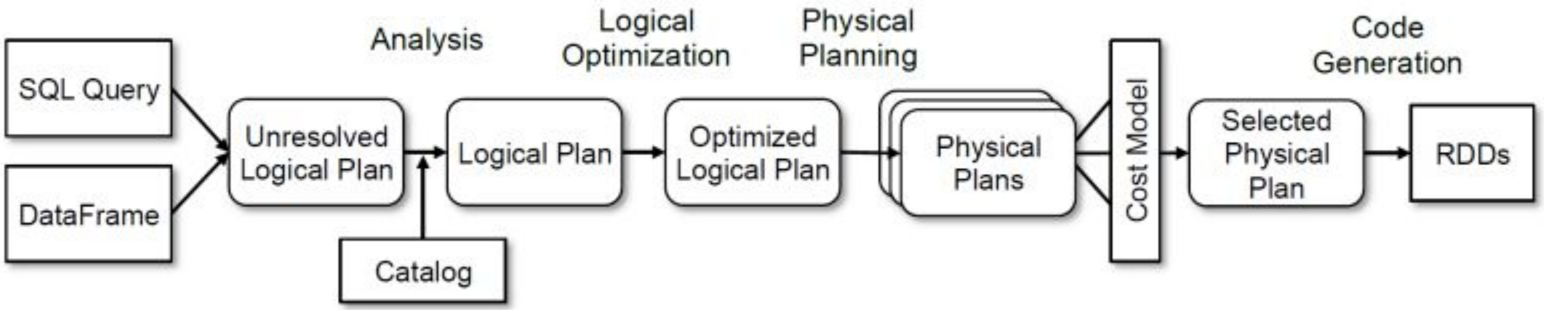
Catalyst: Query Planning



SparkSQL and DataFrame/Dataset

Performance features

1. Whole-stage code generation via Janino
2. Off-heap memory (sun.misc.Unsafe) for caching and certain operations
3. Pushdown of selection, projection, joins into data sources (+ join ordering)



DASK

Overview Dask

- Multi-threaded and distributed operations for arrays, bags, and dataframes
- dask.array: list of numpy n-dim arrays
- dask.dataframe: list of pandas data frames
- dask.bag:unordered list of tuples (second order functions)
- Local and distributed schedulers: threads, processes, YARN, Kubernetes, containers, HPC, and cloud, GPUs

Execution

- Lazy evaluation
- Limitation: requires static size inference
- Triggered via compute()

```
import dask.array as da
x = da.random.random( (10000,10000),
chunks=(1000,1000))
y = x + x.T
y.persist() # cache in memory
z = y[::2, 5000:].mean(axis=1) #colMeans ret
= z.compute() # returns NumPy array
```

Summary and Q&A

Summary and Q&A

- **Summary and Q&A**
  - Motivation and Terminology
  - Data-Parallel Collection Processing
  - Data-Parallel DataFrame Operations
- **Next Lectures**
  - Distributed Stream Processing **[Jan 10]**

Vielen Dank!