DSC-204A: Scalable Data Systems, Fall 2025

11: Streaming Processing, RDD, Spark

Lecturer: Hao Zhang *Scribe: Yue Yu, Hongjie Wang, Yuru Feng, Wenqi Li*

1 Batch Processing and Its Limitations

- **Batch processing** is suitable for latency-insensitive, large-scale data processing tasks.
- The **MapReduce model** expresses computations as a series of map and reduce operations, communicating via disk I/O.
- While MapReduce is scalable and fault-tolerant, it suffers from **low performance** due to frequent disk I/O.

2 Computation vs. I/O: Arithmetic Intensity

2.1 Memory Hierarchy

Data movement through the memory hierarchy—from registers, caches, and DRAM to disks and remote storage—introduces latency and performance loss. Disk I/O is the primary bottleneck in large-scale computation.

2.2 Arithmetic Intensity

$$AI = \frac{\#\text{Compute Operations}}{\#\text{I/O Operations}}$$

A higher arithmetic intensity means better performance since more computation is done per data movement.

2.2.1 Example

Consider the following loop:

```
for (int i = 0; i < n; i++)
    C[i] = A[i] + B[i];
```

Each iteration performs 1 arithmetic operation but involves 2 loads and 1 store, leading to low arithmetic intensity (1/3).

2.3 Loop Fusion

Loop fusion combines multiple loops to improve arithmetic intensity by reducing redundant data accesses. For example:

$$E[i] = D[i] + (A[i] + B[i]) \times C[i]$$

performs more computations per I/O operation than running separate add and multiply loops.

> *Note: The idea of loop fusion is often referred to as **kernel fusion** in modern machine learning and deep learning systems.*

3 MapReduce's Core Problem

The fundamental weakness of MapReduce lies in its **low arithmetic intensity** caused by constant disk reads/writes between iterations.

3.1 Iterative Workloads

Tasks like PageRank, iterative machine learning algorithms, or graph analytics require multiple iterations until convergence. For example, in PageRank: Each iteration depends on the results of the previous one, so the entire graph must be reloaded from disk each round—leading to **low efficiency**.

4 Motivation for Spark

MapReduce's inefficiency in handling iterative and interactive workloads motivated the creation of Spark.

> *Story behind the scene: Spark originated from **Matei Zaharia**'s experiences during the Netflix Prize competition, where he found MapReduce too slow for iterative algorithms. This led to the design of a new framework focusing on in-memory data reuse.*

4.1 Goals of Spark

- Provide a programming model for cluster-scale computations with **significant reuse** of intermediate results.
- Support **iterative** machine learning and graph algorithms.
- Enable interactive data mining by caching large datasets in memory for multiple ad-hoc queries.
- **Avoid inefficiencies** of writing intermediate results to persistent distributed storage.

5 Why Spark Could Succeed

Spark's success relies on three enabling hardware and system trends around 2010–2015.

5.1 Memory Becomes Large and Affordable

- DRAM capacity increased steadily, while price per byte dropped significantly.
- This made it possible to store large intermediate datasets entirely in memory.

5.2 Network Bandwidth Improves Rapidly

- Ethernet bandwidth has been growing by 33–40% per year.
- Faster networks reduce the cost of data shuffling and communication between cluster nodes.

5.3 Fault Tolerance Comparable to MapReduce

- Spark had to maintain MapReduce's robustness while avoiding excessive disk I/O.
- It introduced a lineage-based fault tolerance mechanism to recompute lost data efficiently.

6 The Shifting Memory Hierarchy

With SSDs becoming cheaper than HDDs and network speeds outpacing PCI/SATA throughput, the traditional memory hierarchy shifted upward:

- HDDs are nearly obsolete in modern clusters.
- SSDs serve as the new baseline storage.
- Today's RAM capacity rivals yesterday's SSDs.
- Ethernet is approaching the speed of local I/O buses.

This evolution enabled Spark to perform in-memory distributed computing at scale.

7 Spark and RDDs

7.1 Overview

Resilient Distributed Datasets (RDDs) are introduced by AMPLab at UC Berkeley in paper *Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing*[1].

RDD is Spark's key programming abstraction:

- RDD is **read-only** collection of records (immutable)
- RDDs can only be created by deterministic transformations on data in persistent storage or on existing RDDs.

7.2 RDD Operators: Transformation vs. Action

RDD operators are categorized into two main types based on how they affect the RDD's state and execution flow:

- **Transformation** (Lazy Execution)

[1] https://dl.acm.org/doi/10.5555/2228298.2228301

  - Transformations are operations applied to an RDD that create a new RDD from an existing one.
  - Transformations are **lazy**; they do not compute the result immediately. Instead, they just record the operation (the **lineage**) in a **directed acyclic graph (DAG)**. This allows Spark to optimize the execution plan.
  - Transformation's output is another RDD.

- **Action** (Eager Execution)
  - Actions are operations that trigger the actual computation (or execution) of the entire RDD lineage, sending the data back to the driver program or saving it to an external storage system.
  - Actions are **eager**. When an Action is called, Spark analyzes the DAG, creates a **physical execution plan**, and executes all the pending Transformations necessary to produce the final result.
  - Action's output is a value

See the spark code sample below. By transformation operations form a lineage graph, spark tracks how an RDD is derived from its parent RDDs, which is key to Spark's fault tolerance.

```
// create RDD from file system data
var lines = spark.textFile("hdfs://15418log.txt");

// create RDD using filter() transformation on lines
var mobileViews = lines.filter((x: String) => isMobileClient(x));

// another filter() transformation
var safariViews = mobileViews.filter((x: String) => x.contains("Safari"));

// then count number of elements in RDD via count() action
var numViews = safariViews.count();
```

7.3 RDD and Map Reduce: Comparison

The RDD model offers a high degree of flexibility and expressiveness compared to MapReduce. RDDs offer a large, specialized set of built-in operators (Transformations and Actions). This makes complex data processing logic implementation easier.

7.4 RDD Implementation

The focus of RDD implementation is to avoid storing huge in-memory arrays for every intermediate step.

Should we think of RDDs as simple arrays? The answer is **No**. Representing every intermediate RDD as a full, in-memory array (‘Array‘) would be extremely inefficient and quickly consume all DRAM, as the intermediate files would often be larger than the original data on disk.

8 Spark RDD Lineage, Dependencies, and Fault Tolerance

8.1 RDD Lineage and Symbolic Computation

In Apache Spark, computations on Resilient Distributed Datasets (RDDs) are represented symbolically through a **lineage graph**. Rather than executing transformations immediately, Spark records a sequence of operations to construct a *computational dependency graph* (see Figure 1).

When a user writes a chain of transformations (e.g., `map`, `filter`, `flatMap`), Spark internally builds this graph to track the relationships among intermediate datasets. Execution is deferred until an *action* (such as `collect` or `count`) is invoked.

Once the full lineage is known, Spark's backend performs **loop fusion**: it merges multiple transformations into a single execution loop. This design avoids unnecessary materialization of intermediate results and minimizes disk I/O. The lineage graph thus provides both a global view of computation and an opportunity for code fusion and optimization.

- **Key idea:** Spark constructs a symbolic computation graph first, and executes only once it knows the complete lineage.
- **Benefit:** Enables efficient in-memory computation and avoids repeated reads/writes from disk.

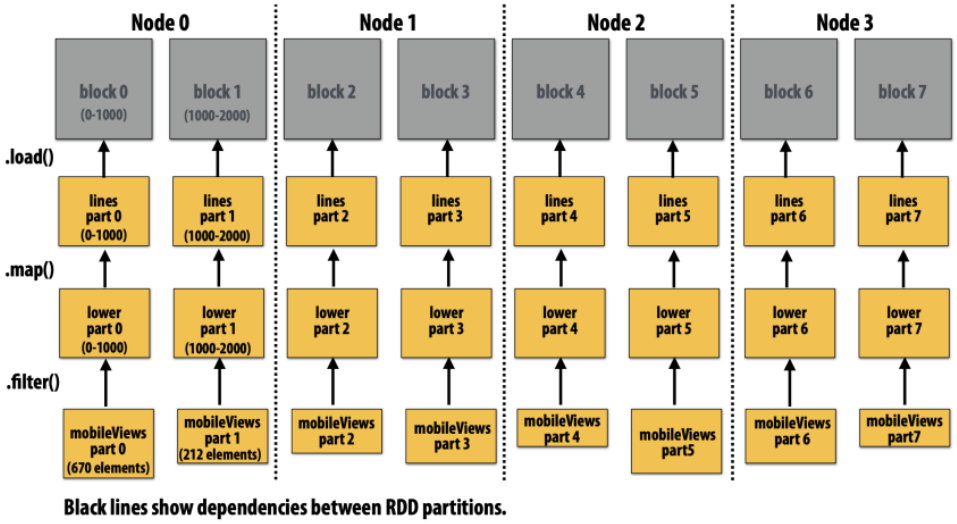

Figure 1: RDD Lineage Graph and Disk Dependencies (Slide 40)

8.2 Narrow vs. Wide Dependencies

Depending on the type of transformation, Spark's lineage can exhibit two types of dependencies:

8.2.1 (a) Narrow Dependencies

Each partition of the child RDD depends on only a single partition of the parent RDD (Figure 1).

- Typical of operations such as `map()` or `filter()`.
- Each partition can be computed independently — this is analogous to **data parallelism**.
- No inter-node communication is required; all computation can be performed locally.

8.2.2 (b) Wide Dependencies

A partition in the child RDD depends on multiple partitions from the parent RDD (Figure 2).

- Caused by operations such as `groupByKey()` or `reduceByKey()`.
- Requires **data shuffling** across the cluster — each node must access results from multiple previous partitions.
- These dependencies create a synchronization barrier, where all parent partitions must complete before the next stage can begin.

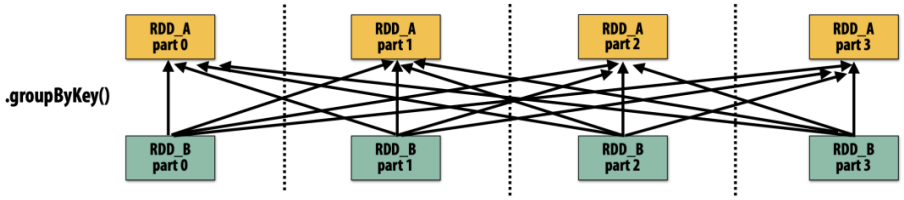

Figure 2: Wide Dependency and Data Shuffling Across Nodes (Slide 43)

Key takeaway: Narrow dependencies allow pipelined, fused computation; wide dependencies require synchronization and inter-node data transfer.

8.3 Stage-Based Scheduling

Spark's execution model divides the lineage graph into **stages** based on dependency boundaries:

1. **Stage 1:** Input data is already materialized in memory — no computation required.
2. **Stage 2:** Spark fuses all **narrow-dependent** transformations into a single execution stage without materializing intermediates.
3. **Stage 3:** Spark executes **wide-dependent** transformations, materializing intermediate results only when necessary.

This staged execution allows Spark to balance efficiency and fault tolerance: transformations are fused where possible, but wide dependency boundaries enforce synchronization to maintain consistency.

Insight: Spark's backend translates lineage graphs into stage-level code execution, minimizing intermediate I/O while preserving deterministic computation.

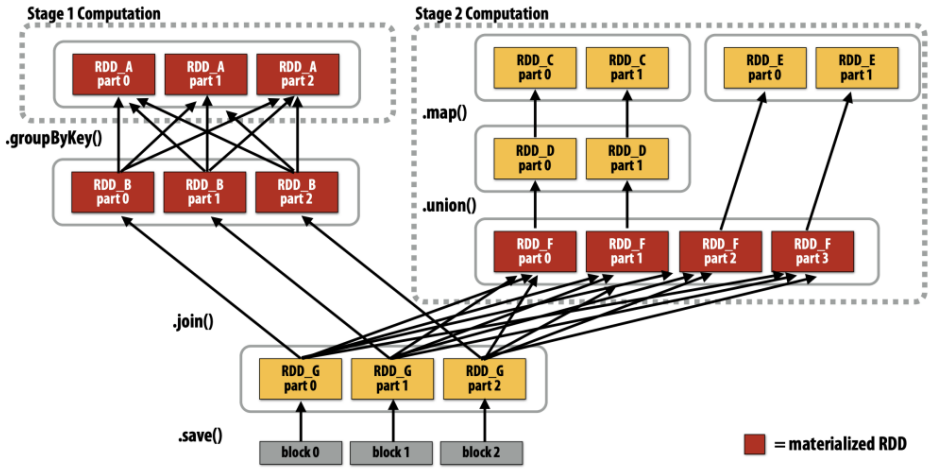

Figure 3: Stage-Based Scheduling and Execution (Slide 45)

8.4 Lineage-Based Fault Tolerance

Beyond optimization, the lineage graph also enables **fault tolerance** without heavy data replication.

- Spark logs the transformation history (the lineage) rather than writing all intermediate data to disk.
- When a node fails, Spark consults the lineage graph to locate the most recent non-failed RDD partition.
- The system then **recomputes lost data** deterministically by replaying the recorded transformations from that point onward.
- This recovery method is both efficient and storage-light compared to full checkpointing.

Summary:

- Lineage graphs provide a unified abstraction for optimization and recovery.
- Logging transformations (rather than data) enables fast recomputation on failure.
- This design allows Spark to achieve high performance and resilience with minimal storage overhead.

9 Spark Performance

Key argument of Spark: It is good at iterative computation, because every intermediate result between iterations is basically materialized in memory, avoiding in disk I/O.

9.1 Spark Improves MapReduce Over

- Easy for programmers because you express your computation by chaining atomic operators
- From much fewer I/O to very improved AI

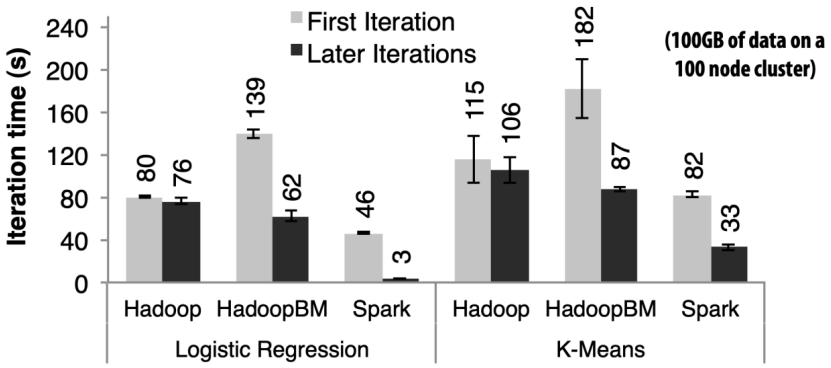

Figure 4: Performance of Spark compared to other systems(Slide 48)

9.2 Spark Cons

1. Debuggability
2. Bulky (Map-reduce is not bulky as it works well if you only have one worker, so now every PL has a "map" function)

The modern Spark ecosystem can support many different tasks like SparkSQL, SparkMLlib, and Spark-GraphX. Complelling feature of Spark, which enables integration/composition of multiple domain-specific frameworks(since all collections implemented under the hood with RDDs and scheduled using scheduler).

Figure 5: Ecosystem of Spark

9.3 Stories of Spark and Databricks

Spark is initially just an open-source project by a few students, the community grows because of advantages over Hadoop and Map-Reduce. However, student were about to graduate and could not commit time to

those projects. In this case, they asked Hortonworks if they wanted to take over but it failed, so eventually they started Databricks. As we know, in Cloudera, data platform company are founded by Hadoop authors. Following is a brief summarizes of this topics:

- Used to be a unicorn / high-profile / high-tech company
- Was beat hard by Databricks / Snowflake
- Went to public 2017, stock price keeps declining. . . , merged with Hortonworks in 2018, went to private in 2021 after being acquired by investment companies.

Compared to Databricks, which has 7 cofounders, they tried to sell SPark but were unsuccessful and struggled for quite a few years. It almost failed during 2018-2020, but when data warehousing and OLAP gradually become a business, it grows with the trend. Now it valued at 100B today and create 7 billionaires.

After Spark, all modern Data/ML Systems follow a similar architecture: a fixed set of operators, a trusted runtime with a small set of pre-loaded implementations.