DSC 232R: Big Data Analytics Using Spark
Winter 2026
Week 2

January 18, 2026

1 Topic: HDFS and The MapReduce Paradigm

1.1 The Distributed File System (HDFS)

1.1.1 Lecture Content

- **The Core Problem:** Standard file systems cannot handle petabytes of data because they rely on a single disk/computer.
- **Google's Solution (GFS/HDFS):**
  1. **Commodity Hardware:** Use thousands of cheap, unreliable computers instead of one expensive supercomputer.
  2. **Chunking:** Files are broken into fixed-size blocks (default 128MB) called "Chunks" or "Splits".
  3. **Redundancy (Replication):** Every chunk is copied to multiple machines (Default Replication Factor = 3).
     - *Why?* If a machine crashes (which happens constantly), the data exists elsewhere.
  4. **Architecture:**
     - **NameNode (Master):** Stores metadata (filename → list of chunk IDs).
     - **DataNode (Worker):** Stores the actual raw bytes.
- **Locality Principle:** "Move computation to the data, not data to the computation."
  - Network bandwidth is the bottleneck. It is faster to send a 5KB program to the data than to pull 1TB of data to the program.

2 Topic: Spark Architecture

2.1 RDDs and Context

2.1.1 Lecture Content

- **SparkContext (The Driver):**
  - The "Brain" of the operation. Resides on the Master node.
  - Does NOT store the data. It stores the *Execution Plan* (Lineage).
  - Coordinates workers (Executors).
- **RDD (Resilient Distributed Dataset):**
  - **Resilient:** Reconstructs lost data automatically using lineage (recomputing steps) rather than replication.
  - **Distributed:** Partitioned across the cluster.
  - **Immutable:** Once created, it cannot be changed. You can only create a *new* RDD from an old one.
- **Spark vs. Hadoop:**
  - **Hadoop MapReduce:** writes to Disk after every step (High Latency).
  - **Spark:** keeps data in Memory (RAM) between steps (Low Latency).

2.2 Lazy Evaluation and Pipelines

2.2.1 Lecture Content

- **Two Types of Operations:**
  1. **Transformations (Lazy):** Definition of a plan. Returns a new RDD.
     - *Examples:* `map`, `filter`, `flatMap`, `sample`.
     - *Behavior:* Spark does NOTHING when these lines are run. It builds a DAG (Directed Acyclic Graph).
  2. **Actions (Eager):** Triggers execution. Returns a result to the Driver.
     - *Examples:* `collect`, `count`, `reduce`, `take`, `saveAsTextFile`.
     - *Behavior:* Spark optimizes the DAG and launches tasks on the cluster.
- **Pipelining (Fusion):**
  - Because of lazy evaluation, Spark can combine multiple steps into one pass.

  - *Example:* `map` + `filter` happens in a single loop over the data, avoiding intermediate memory storage.

- **Caching:**
  - Because RDDs are re-computed by default, branching (using an RDD twice) causes double computation.
  - `rdd.cache()` tells Spark to save the result in RAM after the first computation.

3 Topic: Key-Value Operations (The Danger Zone)

3.1 Pair RDDs

3.1.1 Jupyter Notebook Content

- **Data Structure:** RDDs where elements are Python tuples: `(Key, Value)`.
- **reduceByKey(func):**
  - Groups values by key and applies `func` (e.g., sum).
  - **Architecture Win:** Performs *Map-Side Combination.* It reduces data LOCALLY before sending it over the network.
  - *Result:* Low network traffic, high performance.
- **groupByKey():**
  - Groups values by key but performs NO reduction. Returns `(Key, Iterable)`.
  - **Architecture Fail:** Sends ALL raw data over the network to the reducer.
  - *Result:* High network traffic, likely to cause **Out Of Memory (OOM)** errors on the reducer.
- **The Shuffle:**
  - Operations that require moving data between partitions (by key) cause a "Shuffle".
  - This breaks the pipeline and defines a "Stage Boundary".
  - *Shuffle Ops:* `reduceByKey`, `groupByKey`, `join`, `repartition`.

4 Exam Traps: Danger Zone

- **The "Average" Trap (Algebraic Properties):**
  - You cannot calculate an average using `reduce(lambda x,y: (x+y)/2)`.
  - **Why?** Reduce operations must be **Associative** and **Commutative**.
  - $(a + b)/2 + c$ is not the same as $(a + b + c)/3$.
  - **Solution:** Map to `(sum, count)` tuples first, reduce by adding components element-wise, then divide at the very end.
- **The groupByKey Trap:**
  - Prof will ask: "Why did my cluster crash when I switched from `reduceByKey` to `groupByKey`?"

  - **Answer:** `groupByKey` forces a massive shuffle of all data. `reduceByKey` pre-aggregates on the map side (like a combiner), sending minimal data.

- **The Collect Trap:**
  - `rdd.collect()` brings ALL data to the Driver node.
  - If the RDD is 1TB and the Driver has 16GB RAM, the driver crashes.
  - **Exam Fix:** Use `take(n)` or `saveAsTextFile()` instead.

- **Terminology Shift:**
  - Lecture: "Lazy Evaluation". Exam: "DAG Construction vs. Materialization".
  - Lecture: "Shuffle". Exam: "Wide Dependency".
  - Lecture: "Chunk". Exam: "Partition" or "Split".

5 Practice Quiz

Question 1

The Philosophy of MapReduce

Which statement best describes the ”Data Locality” principle in HDFS/Spark?

a All data is gathered to the Master node for processing.

b Data is randomly shuffled across the network to ensure fairness.

c Code (computation) is sent to the node where the data resides.

d Data is always compressed before processing.

Answer: C

Brief Explanation

- Network bandwidth is the scarcest resource. Moving code (kilobytes) to the data (terabytes) is orders of magnitude faster than moving data to the code.

Question 2

The Math of Reduce

Why does the operation `rdd.reduce(lambda a,b:  a - b)` yield unpredictable results?

a Subtraction is too computationally expensive.

b Subtraction is not Associative.

c Spark does not support negative numbers.

d The lambda syntax is incorrect.

Answer: B

Brief Explanation

- Reduce operations execute in parallel in undefined orders. $(a-b)-c$ is not the same as $a-(b-c)$. Operations must be Associative and Commutative to guarantee deterministic results in a distributed system.

Question 3

Lazy Evaluation

You run the following code on a 10TB dataset:

```
data = sc.textFile("huge_file.txt")
mapped = data.map(lambda x: x.split())
filtered = mapped.filter(lambda x: len(x) > 5)
```

How much time does this take to execute?

a Approx 1 hour (reading 10TB).

b Approx 10 minutes (filtering reduces size).

c Microseconds (almost instant).

d It depends on the number of executors.

Answer: C

Brief Explanation

- `textFile`, `map`, and `filter` are all **Transformations**. Spark is Lazy—it only builds the Execution Plan (DAG) in memory. No data is read until an Action (like `count`) is called.

Question 4

Performance Tuning

You need to count word frequencies. Which operation is more efficient and why?

a `groupByKey().map(sum)`

b `reduceByKey(sum)`

Answer: B

Brief Explanation

- `reduceByKey` performs **Map-Side Combination**. It sums the counts locally on each worker *before* sending data across the network. `groupByKey` shuffles every single word pair, causing massive network congestion.

Question 5

Stage Boundaries

What triggers the creation of a new ”Stage” in a Spark Job?

a Any Transformation.

b Any Action.

c A Shuffle (Wide Dependency).

d A Cache command.

Answer: C

Brief Explanation

- Spark pipelines ”Narrow Dependencies” (map, filter) into a single stage. A ”Shuffle” (reduceByKey, join, repartition) requires data to move between partitions, forcing a hard barrier (Stage Boundary) where all previous tasks must finish before the next begins.

Question 6

HDFS Reliability

If a DataNode crashes, how does HDFS ensure data is not lost?

a It uses RAID 5 on the Master Node.

b It recovers the data from the NameNode's RAM.

c It relies on the 3x Replication Factor (copies exist on other nodes).

d The job simply fails.

Answer: C

Brief Explanation

- HDFS replicates every chunk (default 3 times) across different nodes. If one node dies, the Master simply redirects requests to one of the replicas.

Question 7

Driver vs. Executor

What is the primary risk of running the following command on a production cluster? `result = huge_rdd.collect()`

a It will delete the original data on HDFS.

b It forces the Executors to crash.

c It causes an Out Of Memory (OOM) error on the Driver.

d It runs too slowly because of Python serialization.

Answer: C

Brief Explanation

- `collect()` fetches all data from all distributed partitions and tries to fit it into the memory of the single Driver machine. If the data ¿ Driver RAM, the driver crashes.

Question 8

DAG Visualization

What is the purpose of `rdd.toDebugString()`?

a To print the first 10 rows of data.

b To view the logical Execution Plan (Lineage).

c To check for syntax errors in Python.

d To check the size of the RDD in bytes.

Answer: B

Brief Explanation

- `toDebugString()` outputs the DAG (Directed Acyclic Graph) showing the lineage of transformations and where shuffles/caching will occur.

Question 9

Glom

What does the `glom()` transformation do?

a Deletes empty partitions.

b Coalesces all partitions into one.

c Transforms each partition into a list (array) of elements.

d Sorts the data globally.

Answer: C

Brief Explanation

- `glom()` maps each partition to a single array containing all elements of that partition. It is useful for efficient batch operations or checking partition skew (e.g., `glom().map(len).collect()`).

Question 10

Immutability

You want to add 5 to every element in an RDD ‘A‘. You run ‘A.map(lambda x: x+5)‘. What happens to ‘A‘?

a ‘A‘ is modified in place.

b ‘A‘ is deleted and replaced by the new result.

c ‘A‘ remains unchanged; a new RDD is returned.

d It depends on whether ‘A‘ is cached.

Answer: C

Brief Explanation

- RDDs are **Immutable**. You cannot change an RDD. Transformations always return a *pointer* to a new RDD that depends on the old one.