Week 07: Column-Oriented Storage
(Full Detailed Notes)

Introduction

Classroom Instructions:

- Please silence and put away cell phones.
- The slides are for note-taking support only, not a substitute for attending lectures.

Focus of today's lecture:

- Using Apache Spark (for distributed computation)
- Understanding column-oriented storage in big data
- Learning about Dremel and Parquet (two systems for handling structured/nested data at scale)

Core Message:
Even in a world where parallelism is important, data structures matter a lot! Organizing your data wisely is crucial for efficiency.

Part 1: Spark Review — Partitions and Dependencies

Narrow Dependencies

Definition:
Each partition of the parent RDD (Resilient Distributed Dataset) is used by at most one child RDD partition.

Properties:

- Minimal data movement (localized).
- Low communication overhead.
- Easy to pipeline operations.
- Easy failure recovery because no dependency on multiple partitions.

Result:
Fast operations because no expensive shuffle phase.

Wide Dependencies

Definition:
A partition of the parent RDD contributes to multiple child RDD partitions.

Properties:

- Requires key shuffle (expensive, time-consuming).
- High communication cost across the cluster.

- High latency.
- Difficult to pipeline.
- Harder to recover from failures.

Result:
Slow operations because data has to move between machines.

Part 2: The Problem of Wide Dependencies (and Co-Partitioning)

Example Scenario:

Three datasets:

Students (s)	Majors (m)	Academic Records (ar)
sID	mID	sID
Name	Name	Course
Major	School	Grade
Full-Time (FT) status		Credits

Goal:
Suppose we want to check if full-time students take enough credits. We need to join the Students and Academic Records tables on sID.

Problem:
A join creates a wide dependency:

- Students data is spread across partitions.
- Academic Records data is spread differently.
- Need a shuffle to match them = slow.

Solution: Co-Partitioning

Idea:
Pre-arrange (pre-shuffle) data so that records with the same join key (sID) land in the same partition. Done before the expensive operation.

In Spark:

```
s.repartition(3, "sID")
ar.repartition(3, "sID")
```

This way, same student IDs are co-located.

Example:

- Student with sID=1 goes to partition 1
- Student with sID=2 goes to partition 2
- and so on...

Important Notes:

One-time shuffle:
Co-partitioning does require a shuffle up-front, but it avoids repeated future shuffles for frequent access patterns (e.g., GPA calculation).

Key-specific Optimization:
If you co-partition by sID, operations based on sID become fast. But if you need to group by Major later, you need to repartition again.

How Spark Knows?
Spark tracks partitioning info using internal metadata and Catalyst Optimizer (written in Scala).

Types of Co-partitioning:

- `repartition()` — temporary, in-memory.
- `bucketBy()` — persistent, stored on disk.

Part 3: Using Apache Spark

Quick History:

- Developed at UC Berkeley in 2009.
- Open-sourced in 2010.
- Published (paper) in 2012.
- Released 1.0 in 2013.

Key Components:

- RDDs (Resilient Distributed Datasets) — distributed memory objects.
- Integration with Hadoop:
  - HDFS for storage.
  - YARN for scheduling jobs.
- Written in Scala, but APIs exist for:
  - Python (PySpark)
  - R
  - Java
  - MATLAB (via MDCS)

Architecture:

Driver and Sessions:

- Driver process:
  - Runs on head/login node.
- Session:
  - Connects user code to the Spark cluster.

Illustration (in words):

- Driver starts a session.
- Session coordinates distributed workers (executors).
- Cluster Manager (like YARN) allocates resources.

Spark DataFrame API

Problem with RDDs:

- Flexible but cumbersome for ad-hoc tasks.

Solution:

- DataFrames offer a structured, easy-to-use abstraction.
- Inspired by pandas (Python) and R DataFrames.

Fun Fact:
Internally, DataFrames are built on top of RDDs.

DataFrames vs RDDs:

Aspect	DataFrames	RDDs
Structure	Schema-defined columns	Raw distributed objects
Optimized?	Yes, through Catalyst	No
User Experience	Easier (pandas-like)	Harder (manual transforms)

In Spark 2.x and beyond, DataFrames are primary interface.

Spark SQL

You can query DataFrames using SQL!

Example:

```
df.createOrReplaceTempView('students')
spark.sql("SELECT Major, AVG(Credits) FROM students GROUP BY Major").show()
```

Equivalent without SQL:

```
df.groupBy('Major').avg('Credits').show()
```

Best Practice:
Always run `.explain()` to see the execution plan.

Caution:

- Avoid `.collect()` on huge data — dangerous (might crash the driver).
- Use `.take(n)`, `.save()`, `.show()` instead.

Part 4: Column-Oriented Storage

Why Care About Storage Layout?

Problem:
Storage bandwidth and latency have become bottlenecks compared to CPU speeds.

Goals:

- Reduce number of bytes transferred.
- Optimize memory access patterns.

Row-Oriented Storage (e.g., CSV)

Example:

```
id, name, mass
1, T.Rex, 8000
2, Stegosaurus, 4000
3, Ankylosaurus, 4000
```

Easy for humans.
Terrible if you want:

- Only the mass column.
- Only the 1000th record.
- Variable-length rows → no easy indexing → full scan needed.

Problem:

- Time Complexity: O(n)
- Slow random access.

Column-Oriented Storage

Key Change:
Store each column separately.

Example:

```
id: [1, 2, 3]
name: ["T.Rex", "Stegosaurus", "Ankylosaurus"]
mass: [8000, 4000, 4000]
```

Benefits:

- Accessing a single column is fast.
- Easier to compress (same data types).
- Sequential disk reads → better cache utilization.

Compression in Column Stores

Why compression matters:

- Saves space.
- Saves time (smaller data transferred).
- Sometimes enables operations on compressed data directly.

Part 5: Compression Techniques

Compression	Description	Example
Dictionary Encoding	Map frequent values to small integers	"Pop" → 0, "Rock" → 1
Bit Packing	Store small integers tightly without gaps	[0,1,0,2,1] packed
Run-Length Encoding (RLE)	Store value + repetition count	[8000, 1], [4000, 4]
Frame of Reference	Store base + offsets	1004,1005,1006 → base 1000, offsets [4,5,6]
Delta Encoding	Store first value + successive differences	1004,1005 → 1004, +1
Huffman Coding	Variable length coding based on frequency	Common values = shorter codes

Choosing compression depends on data patterns and access goals.

Part 6: Structured Data (Trees) — Dremel

Challenge with Nested Data:

Real-world data (e.g., web pages, logs) is hierarchical and nested, not tabular.

Example:

- Document ID
- Links (backward, forward)
- Names (with Language info)

Dremel's Core Ideas:

Flatten tree-structured records into a table.

Keep track of:

- Repetition Level (r): Where a repeated field occurred.
- Definition Level (d): How deeply nested the field is, counting missing optional fields.

Result:

Hierarchical data becomes queryable like columns!

Part 7: From Dremel to Parquet

Parquet (2013) = Open-source implementation of Dremel ideas.

Stores structured, nested data efficiently in columnar format.

Parquet File Layout:

- File
  - Row Group 0
    - Column Pages
  - Row Group 1

        * Column Pages
    - Footer (Metadata)

Metadata tells Spark:

- Which row groups to scan or skip.
- How data is compressed.

Part 8: Benefits and Costs of Parquet

Pros:

- Cross-language (Python, Java, C++) support.
- Only necessary columns decoded.
- Built for large distributed file systems (like HDFS).

Cons:

- Complex format (binary, not human-readable).
- Requires proper tooling to read/write.

Final Wrap-up

- Column stores are better for attribute-heavy analysis.
- Dremel lets us flatten structured/nested records.
- Parquet is the de facto standard for big data columnar storage.
- Even in big data, good data structure choices are critical.