1 BIG DATA TIMELINE* ~ Main Big Data architectures & how Spark contributes/ advantages to each

RELATIONAL DBs
- Structured data storage & SQL.
- Spark: Handles massive distributed datasets beyond single server limitations of traditional RDBMS (while supporting SQL via Spark).

GOOGLE FILE SYSTEM / MAPREDUCE
- Distributed file storage & parallel processing.
- Spark: Faster via in-memory processing vs. MapReduce's disk-based operations.

CASSANDRA
- NoSQL database for massive scale (linear scalability)
- Spark: Better for analytics & complex queries (Cassandra great for simple key-value operations)

SPARK
- In-memory distributed computing.
- Solves speed limitations of Hadoop MapReduce and enabled stream processing.

DATABRICKS
- Unified analytics platform, integrated environment for data engineering & analytics.
- Spark: complementary (Spark is Databricks' core processing engine).

KUBERNETES
- Container orchestration, manages distributed data applications at scale.
- Spark: Purpose-built for data processing vs. general container orchestration, but often used together.

DATA LAKEHOUSES
- Data warehouses + lake capabilities (ie. Databricks, Snowflake). Unified structured/unstructured data processing.
- Spark: computational backbone for these platforms.

DATA MESH
- Domain-oriented, distributed data architecture.
- Spark: can serve as the standardized processing layers across different domains in the data mesh architecture.

1970s

DATA WAREHOUSES
- Centralized data storage for analytics.
- Spark: More flexible schema handling, can process unstructured data.

1996

2003

HADOOP
- Open-source implementation of GFS/MapReduce.
- Spark: Simpler API, interactive queries & in-memory processing vs complex MapReduce.

2006

2008

HBASE
- Columnar NoSQL database on Hadoop.
- Real time random access to large datasets.
- Spark: More flexible data processing capabilities (but HBase better for random access)

2009

2010

KAFKA
- Distributed streaming platform. Real time data ingestion & processing at scale.
- Spark: Structured streaming provides higher-level abstractions & built-in processing capabilities.

2011

2012

DOCKER
- Container platform, management of distributed data applications at scale.
- Spark: Purpose-built for data processing vs. general container orchestration. Often used together.

2014

2015

DELTA LAKE
- ACID transactions on data lakes, solved reliability issues with traditional data lakes.
- Spark: Delta lake built on top of Spark (ACID processing).

2017

2019 - Today

VECTOR DBs
- Efficient storage, similarity search for AI embeddings.
- Spark: good for large-scale data processing & transformation (but Vector DBs superior for similarity search.

2020 - Today

2022 - Today

Albert Campillo

*additional, not covered in-depth in the lectures (for context setting purposes)

APACHE SPARK FUNDAMENTALS

2 SPARK : Pros & Cons

- More effective RAM handling vs. prior distributed compute technologies (HIVE, Java MR, ...)

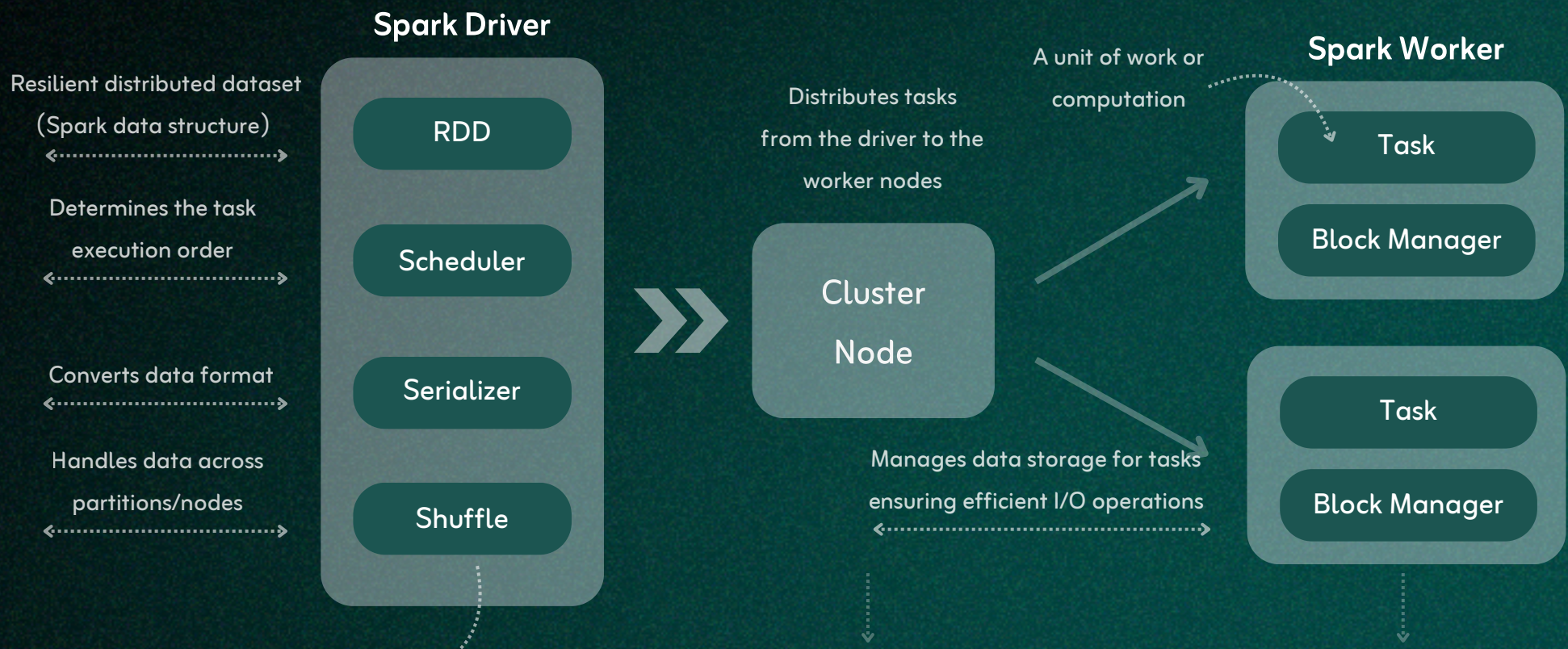
  ie. GROUP BY computation
  - w/o Spark (Hive/MapReduce): read & writen on disk (resilient but slow)
  - w/ Spark: only writes in disk for operations that have not enough memory (spilling to disk)

  Use RAM when possible (effective & fast)
- Storage agnostic (relational db, data lake, file, MongoDB,...), allows storage/compute decoupling
- Massive support community

- If no one else in your organization knows Spark (not immune to bus factor)

  Bus factor: measurement of the risk resulting from information/capabilities not being shared.
  Risk you are taking if key people (ie. only person in the organization that knows Spark) were hit by a bus.
- Your organization already leverages something else (BigQuery, Snowflake,...). Pipeline homogeneity in one technology always preferred!

Apache Spark Driver Execution Flow*



The 'brain' of the operation

- **Application Logic:** hosts the program defining your transformations/actions
- **RDD Management:** tracks data transformation/ maintains fault tolerance
- **Scheduler:** optimize task execution
- **Suffles & Serialization:** moves data efficiently/ converts data into transferrable formats

The 'bridge' of the operation

- Allocates resources to Spark applications
- Communicates task details between Driver & Workers

The 'unsung heroes' of the operation

- **Executing tasks:** each worker processes a data partition
- **Block manager:** manages intermediate data (efficient storage/ retrieval)
- **Replication:** enhances fault tolerance by duplicating key data blocks
- **Task execution:** worker's resources
- **Memory Management:** intermediate data storage (computation)

Albert Campillo | * Muhammad Usman Shahbaz

Apache Spark Driver Execution Flow

THE CHICAGO BULLS' ANALOGY

1
THE PLAN
The Triangle Offense (Play)

THE OVERALL STRATEGY THAT NEEDS TO BE EXECUTED

Phil Jackson's triangle offense is a complex system requiring precise player movement & decision-making

2
THE DRIVER
Phil Jackson (Coach)

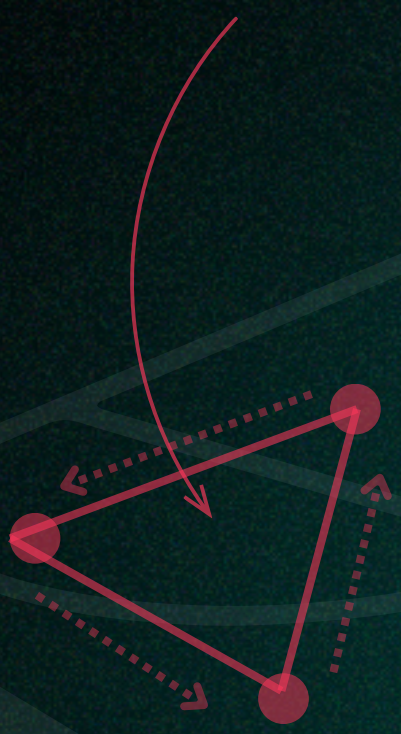
CENTRAL COORDINATOR THAT READS THE PLAN & ORCHESTRATES EXECUTION

Phil Jackson reads the game & makes strategic decisions on when to run plays

3
THE CLUSTER MANAGER
Tex Winter (Assistant Coach)

RESOURCE ALLOCATOR & TASK DISTRIBUTOR

Tex Winter helps Phil Jackson manage player's rotation & communicate plays



5
DATA MANAGEMENT
The Court

HOW DATA MOVES BETWEEN OPERATIONS

Ball movement in triangle offense, creating spacing & opportunities

Steve Kerr (3-point shooting)
Executor optimized for specific operations

Michael Jordan & Scottie Pippen (Scoring)
Executor optimized for specific operations

Dennis Rodman (Rebounding)
Specialized executor for specific tasks

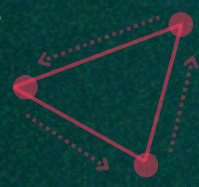
4
THE EXECUTORS
The Players

WORKERS THAT PERFORM THE TASKS

APACHE SPARK FUNDAMENTALS

THE PLAN

- The transformation described (ie. a dataframe/ SparkSQL/ dataset API code...) in either Python, Scala, SQL, R.
- Executed under lazy evaluation (only when it needs to happen) to either
  - write output
  - collect call to determine next set of transformations (part of the plan depends on the data itself).



THE DRIVER

- Reads the plan, identifies a write output or collect call. Decides on:
  - When to start the job and/or being lazy
  - How to JOIN datasets (performance trade-offs)
  - How much parallelism each step needs

Spark settings

~10 different driver settings. Only tweak these 2 (if necessary)

spark.driver.memory

- Memory driver has to process job (default set: -2gb; up to 16gb)
- Cases for memory increase
  - Complex jobs: many steps and/or plans
  - Jobs using dataframe.collect() (the data the job processes changes the plan) - Bad practice!

spark.driver.memoryOverheadFactor

- Extra memory the driver needs for complex jobs & taken from Heap memory (memory Java needs to run)
- How much extra memory? ~ 10% of driver non-heap.

THE EXECUTORS

- Execute the plan passed by the driver

Spark settings

spark.executor.memory

- Memory executors gets (default set: -2gb; up to 16gb)
- Spill to disk issue when memory run is low can lead to slow or break the job.

  Watchout: cost of engineering time > extra cloud storage
- Ideal memory setup? Test run at different levels (ie. 2/4/6gb) for several days and choose the smaller one that runs all the time.

spark.executor.cores

- Number of tasks per machine (default set: 4; up to 6).
- Parallelism & execution speed increases at the increase of tasks, but can cause out of memory issues (higher probability of skew).

spark.executor.memoryOverheadFactor

- Extra memory executors need for complex jobs (handling many UDFs)
- UDFs handle JOIN or UNION operations (memory intensive)
- How much extra memory? ~ 10% of non-heap tasks.