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, ...)

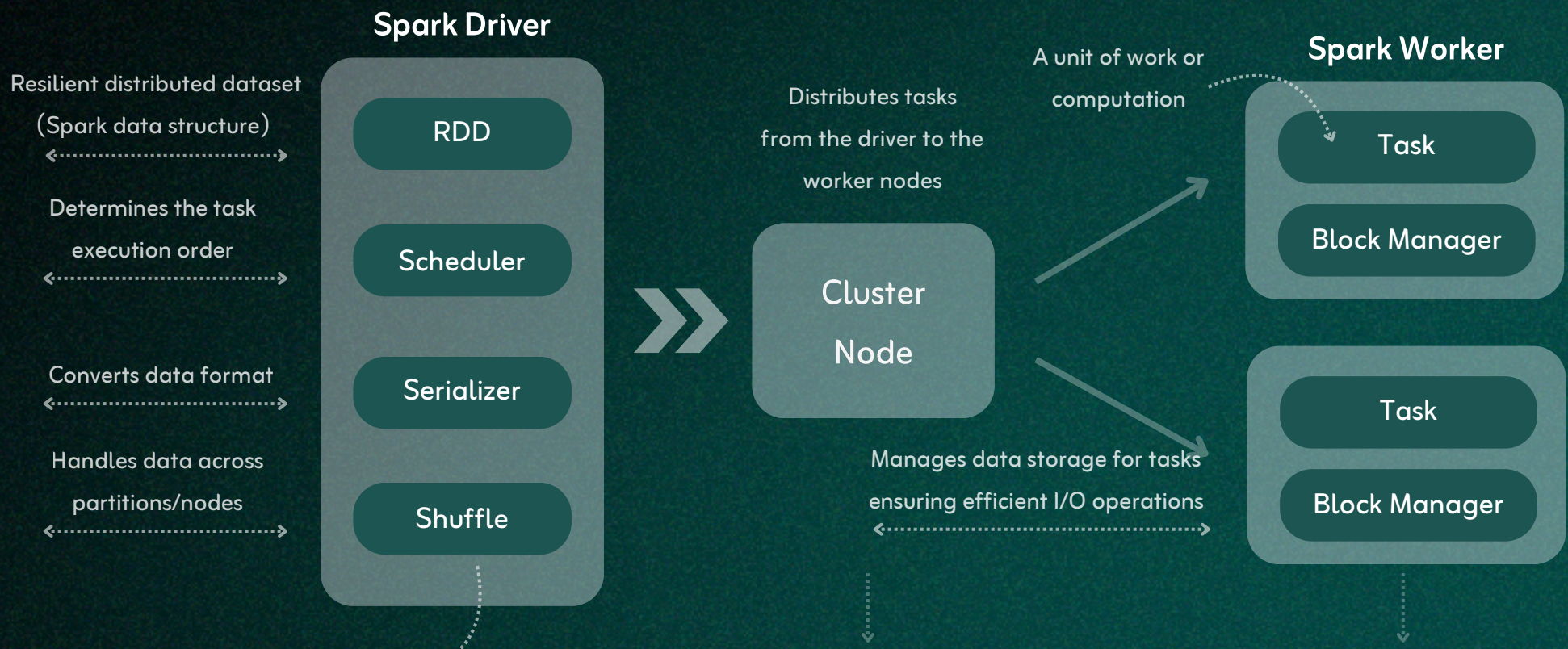
  ie. GROUP BY computation
  - w/o Spark (Hive/MapReduce): read & writen on disk (resilient but slow)
  - w/ Spark: only writes in disk for operations that have not enough memory (spilling to disk)

  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)

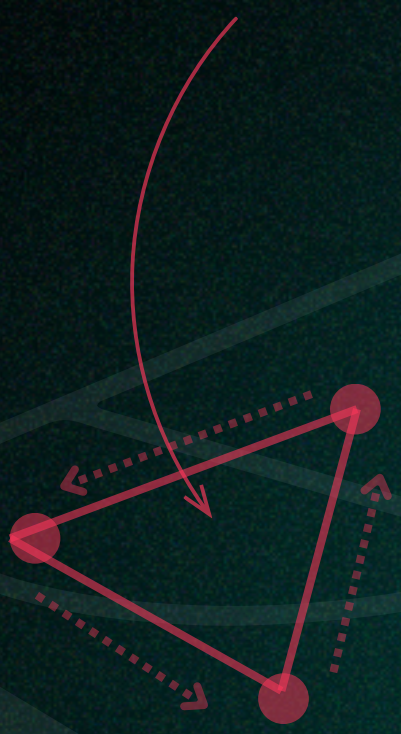
CENTRAL COORDINATOR THAT READS THE PLAN & ORCHESTRATES EXECUTION

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

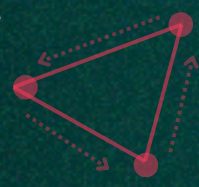
4
THE EXECUTORS
The Players

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.

📌 TYPES OF JOINS IN SPARK

1 SHUFFLE SHORT-MERGE JOIN

Default JOIN strategy in Spark 2.3+ that performs shuffling & sorting of both datasets before merging

When to use ✅

- Large datasets on both sides of the join (>10Gb).
- No significant memory constraints.
- Data distribution across partition is not heavily skewed.

When not to use: ❌

- One side of the join is significantly smaller.
- Need for fast join performance.
- In in-memory constrained environments where shuffling can cause problems.

2 BROADCAST HASH JOIN

Broadcasts the smaller dataset to all executors, eliminating the need for shuffling

When to use ✅

- Left side of join is small (<10Gb)
- Have sufficient memory in your executors.
- Avoid shuffle operations for better performance.

When not to use ❌

- Broadcast side exceeds spark.sql.autoBroadcastJoinThreshold (default 10MB)
- Both datasets are large.
- Limited executor memory.

3 BUCKET JOIN

Efficient join that uses pre-bucketed data to avoid shuffling during the join operation

When to use ✅

- Data already bucketed by join key.
- Buckets in both tables are multiples of each other (powers of twos)
- Perform the same joins frequently
- Avoid shuffle operations but can't use broadcast.

When not to use ❌

- Data is not pre-bucketed.
- Highly skewed bucket sizes.
- The overhead of maintaining bucketed tables isn't worth the join performance gain.

SHUFFLE

- Happens when doing a wide transformation (GROUP BY, JOIN...)
- Spark brings together data that is related but currently resides across different nodes in the cluster

Shuffle Partitions & Parallelism linked!

Paralellism: the application's ability to distribute work across executor cores (how many tasks can be executed simultaneously across your Spark cluster).

🔗 Shuffle partitions determine how data is redistributed across the cluster during operations like JOINs, GROUP BYs, or repartition.

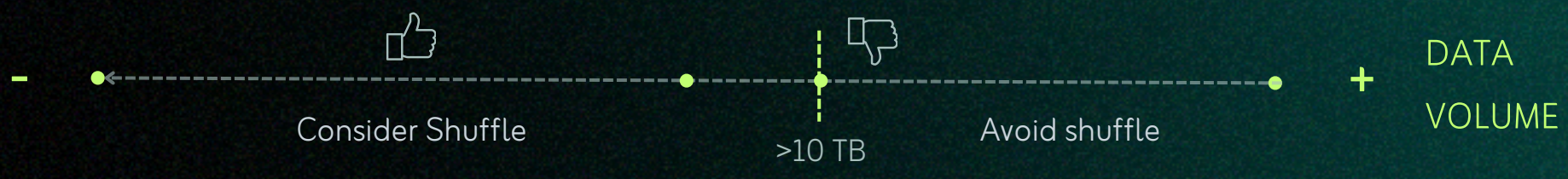
Configurations

- Primary configuration: `spark.sql.shuffle.partitions`
- Secondary configuration: `spark.default.parallelism` (mainly for RDD API ~ avoid usage 99% of cases. Use higher-level APIs - DataFrame, Dataset,...-).

How they are connected:

- Each shuffle partition becomes a task.
- Number of shuffle partitions = Maximum potential parallel tasks (default: 200 partitions).

SHUFFLE: GOOD OR BAD?



Handling Network requests at NETFLIX

Context: Netflix handles network requests, sent to microservice apps (how these apps talk to each other). This is a processing volume of ~100Tb/ hour.

Task: Identify the location of every network request & map/ join every one to a microservice app (thousands in Netflix's architecture)

Initial solution: broadcast join by using a lookup table containing all IP addresses (several Gb) and ship all network requests to microservice apps.

Problem: the service upgrade from IPV4 to IPV6 (>1000x more IP addresses vs. IPV4... a much bigger search space)
Broadcast join impossible & shuffle didn't work.
Bucketing 100Tb/hr? Too expensive.

End solution: upstream approach. Every microservice owner log their app when a network request is received (putting the app in the data avoids JOIN operation). No Spark optimization needed.

Feature generation for notifications at

Context: joining two large tables (10Tb & 50Tb, respectively) at notification level.

Task: Identify a solution that would allow combining the datasets with minimum performance & compute impact.

Initial solution: shuffle merge join, but at a 30% total compute expense (temporary solution).

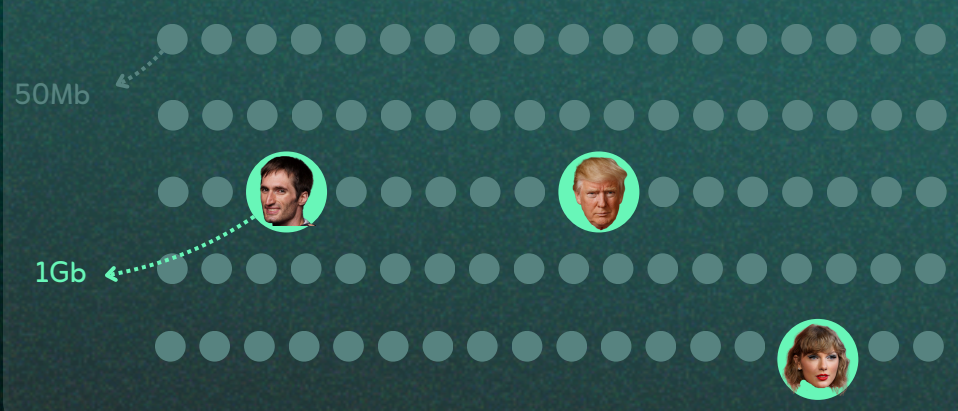
End solution: bucket the tables on user ID & do the JOIN w/o shuffle.
When things are bucketed, files are already have guarantee the data based on that id is in that file.
Even if the two tables don't have same number of buckets can still make a bucket join assuming there are multiples of each other. In this case image a table w/ 1 bucket, another table w/ 2 buckets.
Files 1 & 2 would have all the data in file 3, all would line up because there're multiple of each other

SHUFFLE & SKEW

Skew: occurs when certain partitions contain significantly more data than others during shuffle operation (imbalance can severely impact performance.

- Causes:
  - Insufficient number of partitions
  - Natural data distribution patterns
- Consequences
  - Single executor becomes overwhelmed
  - Pipeline appears 99% complete but fails
  - Overall job performance suffers

Example: Process twitter data for 200 accounts. Use a cluster with 200 partitions, (one per user ID). There might be some accounts with 1000x more data than regular accounts.
Issues for partitions handling large amounts of data.



Albert Campillo

Repost

APACHE SPARK FUNDAMENTALS

Dealing with Skew

1. Adaptive Query Execution (Spark 3+)
   - Enable with `spark.sql.adaptive.enabled` = True
   - Spark automatically adjusts execution plans based on runtime statistics

2. Salting Strategy (Pre-Spark 3)
   - Add random numbers to skewed keys
   - Two-phase aggregation process:
     - Aggregate by salted key (distributes load)
     - Final aggregation after removing salt
   - Important: Break down complex aggregations (AVG into SUM/COUNT)

3. Handling Skewed Joins For cases where salting isn't applicable:
   - Identify and isolate outliers (high-volume keys)
   - Process strategies:
     - Filter out outliers for separate processing
     - Split pipeline into normal and outlier paths
     - Process outliers in dedicated operations

Best Practices

- Use Adaptive Query Execution when available
- Consider salting for GROUP BY operations
- For joins with known outliers, split processing
- Balance complexity of solution vs performance gain

📌 Managed vs. Unmanaged Spark

	Managed Spark (ie. Databricks)	Unmanaged Spark (ie. Big Tech)
Notebooks?	✓	POC Only
How to test the job?	Run the notebook	spark-submit from CLI
Version control	Git or Notebook versioning	Git

How can spark read data

- From the lake
  - Delta lake, Apache Iceberg, Hive Metastore
- From an RDBMS
  - Postgres, Oracle, ...
- From an API
  - Make Rest call & turn into data (careful: this usually happens on the driver - out of memory issue if call is large. Solution: parallelize!)
- From a flat file
  - CSV, JSON