📌 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.

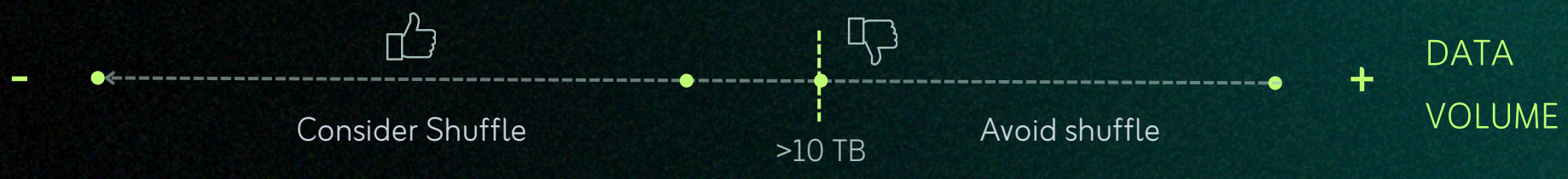
Configurations

- 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).

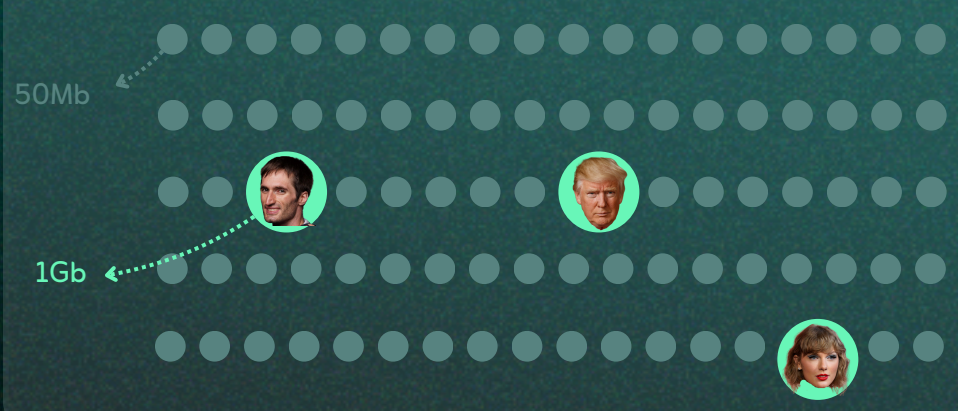
End solution: bucket the tables on user ID & do the JOIN w/o shuffle.
When things are bucketed, files are already have guarantee the data based on that id is in that file.
Even if the two tables don't have same number of buckets can still make a bucket join assuming there are multiples of each other. 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