# APACHE SPARK: ADVANCED CONCEPTS

## 1 DEVELOPMENT ENVIRONMENTS

### 1 SPARK SERVER

approach

Characteristics

- Job submission via CLI with Java class containing run method.
- Fresh environment for each run (automatic uncaching).
- Similar to production environment.

Technical considerations:

- More accurate performance testing.
- Better resource cleanup.
- Supports proper deployment workflows.
- Enables CI/CD integration.
- Good for production-like testing.

### 2 SPARK NOTEBOOKS

NETFLIX approach

Characteristics

- Interactive development environment.
- Persistent Spark session until explicit termination.
- Requires explicit unpersist() calls.

Technical considerations:

- Faster for development and prototyping.
- Cache behavior differs from production.
- Risk of memory leaks without proper cache. management.
- Better for exploratory data analysis.
- Good for collaborative development.

## 2 CACHING & TEMPORAL VIEWS

### CACHE

Temporary storage of computed data in memory and/or disk. Allows to reuse the same data w/o recomputing it.

Storage Levels

#### 1 MEMORY ONLY

Features

- Primary choice for most scenarios.
- Fastest access time.
- Limited by available RAM.

#### 2 DISK ONLY

Features

- Used when memory is insufficient.
- Slower than memory caching.
- Consider staging tables instead.
- Similar performance to materialized view.

#### 3 MEMORY & DISK

Features

- Hybrid approach.
- Spills to disk when memory is full .
- Not always optimal for performance.

Rule of Thumb for storage

1. Memory caching when possible always.
2. Use staging tables instead of disk caching.
3. For edge cases with large, frequently reused results
   - Staging tables over disk caching.
   - Break down jobs into manageable tasks (faster processing).

### TEMPORARY VIEW

A named reference to DataFrame in Spark ('virtual table').

- No physical storage (query recomputed each time).
- Exists within a specific Spark session only (lost when session ends).
- Created using `createTempView()` or `createOrReplaceTempView()`.

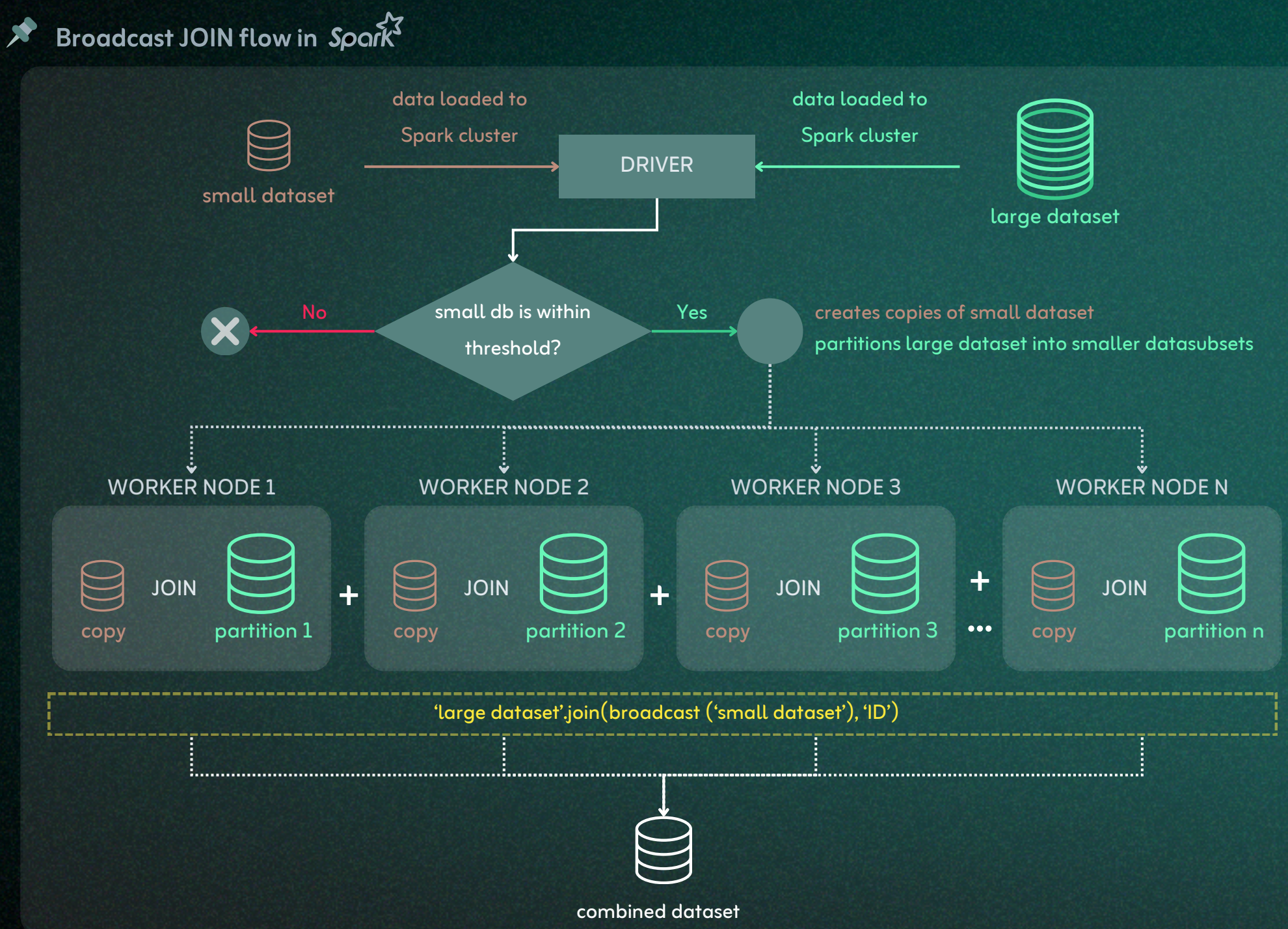
Albert Campillo

Repost

## 3 CACHING VS. BROADCAST JOIN

| | CACHING | BROADCAST JOIN |
|---|---|---|
| Purpose | Stores pre-computed values for reuse across multiple operations. | Optimizes join operations for scenarios with one small & one large dataset. |
| Data distribution | Maintains data partition across the cluster. | Copies the entire small dataset to all executors in the cluster. |
| Memory management | Data is distributed across all nodes (memory efficient). Handles large datasets as data is partitioned. | Highly efficient for small-large table joins (no shuffle). Memory threshold default to 10Mb (can be either changed manually or dataset wrapped into broadcast()). |
| Use cases | Large datasets that need multiple transformations. Partitioned data that needs to remain partitioned. Iterative tasks & repeated queries on same dataset. | Joining a large dataset with a small lookup table. Smaller dataset can fit into the executor memory (<2Gb). Network bandwidth is not a bottleneck for broadcasting. Perform joins without shuffling. |

### Broadcast JOIN flow in Spark



# APACHE SPARK: ADVANCED CONCEPTS

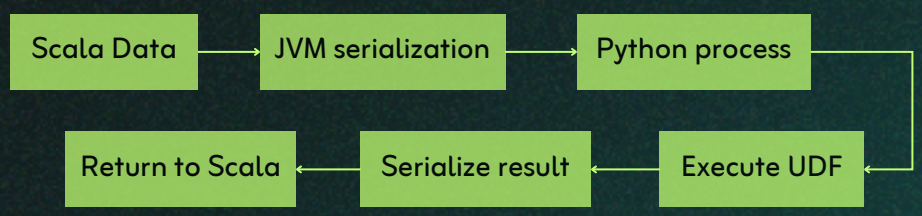
## 4 UDFs / UDAFs

- UDF (User Defined Function): a way to implement custom column transformations & complex data processing logic within Spark DataFrame operations
- UDAF (User Defined Aggregation Function): more specialized than regular UDFs, designed specifically for aggregation operations.

## 5 PySpark vs. ScalaSpark

### PYSPARK

Complex UDF workflow




Considerations:

- Python-user friendly.
- Broader community/ ecosystem.
- Improved by Apache Arrow but still performance issues.
- Performance overhead due to serialization.

Recent Improvements:

- Apache Arrow integration improved PySpark UDF performance.
- Better alignment between PySpark & Scala Spark UDFs through optimization.
- Dataset API benefits:
  - Enables pure Scala functional programming
  - Eliminates need for traditional UDFs
  - Provides type-safe operations
  - Better performance characteristics

### SCALASPARK

Considerations

Pros

- Direct JVM execution.
- More performant UDAFs.
- Native Dataset API access.
- Better performance (no serialization overhead).

Cons

- Not free (commercial licensing).
- Steep learning curve.

Zach's recommendations

Primary recommendation

- PySpark over Scala (more job opportunities)
- Scala if coming from Java background

Language transition considerations

- Java to Scala: easier transition (both JVM languages)
- Python to Scala: tough transition (interpreted vs. compiled languages)

Performance optimization tips

- Use Dataset API in Scala when possible
- Leverage Apache Arrow optimizations in PySpark
- Consider UDAFs for aggregation operations in Scala

Albert Campillo

Repost

# APACHE SPARK: ADVANCED CONCEPTS

## 6 The API continuum

### Spark SQL

- Ideal for multi-user collaboration (data analysts/scientists).
- Fast prototyping/ experimentation.
- Most flexible for rapid changes
- Lowest entry barrier (SQL only).
- Simple null returns for null encounters (less strict nullability controls).

### Spark DataFrame

- Ideal for hardened PySpark pipelines with minimal change requirements.
- Enables code modularization.
- Better testeability with function separation.
- Flexibility-structure balanced.
- Simple null returns for null encounters (less strict nullability controls).

### Scala Dataset

- Ideal for strong software engineering practices & enterprise solutions.
- Enhanced schema handling
- Superior unit & integration testing capabilities.
- Easy mock data generation.
- Built-in type safety.
- Explicit null handling: required nullability declarations (pipeline fails if nullability rules violated).

## 7 Parquet

Parquet: columnar file format, run-length encoding, benefits from parallel processing more efficient querying.

Run-length encoding benefits:

- Leverages data recency & primary patterns.
- Reduces cloud storage costs & network traffic, better compression ratios, eliminates cross-partition data movement

Sorting recommendations:

- DO: `.sortWithinPartitions` (parallelizable operation, maintains good data distribution, sorts locally, cost effective, no expensive shuffle operations).
- DONT: `Global.Sort()` (very slow performance; resource intensive; creates additional shuffle step; all data forced to one single executor; requires strict order of maintenance; computationally expensive)

Technical implementation

1. Partition level processing
2. Performance optimization
3. Storage efficiency

## 8 Spark Tuning

Memory Strategy

1. Executor: don't arbitrarily set to 16Gb (configure based on workload requirements).
2. Driver: bump only when needed.

Shuffle Partitions:

1. Default: 200 partitions (~100 Mb/partition).
2. Optimized: test w/ multiple partitions (1000, 2000, 3000,...), measure performance metrics, incrementally adjust parameters. Check job type:
   - I/O heavy job? Focus on partition optimization
   - Memory heavy job? Balance executor memory
   - Network heavy job? Prioritize network topology & data locality

AQE (Adaptative Query Execution)

- Optimizes for skewed datasets.
- Automatically adjusts query plans based on runtime stats
- Don't enable preemptively (let Spark handle skew optimization).