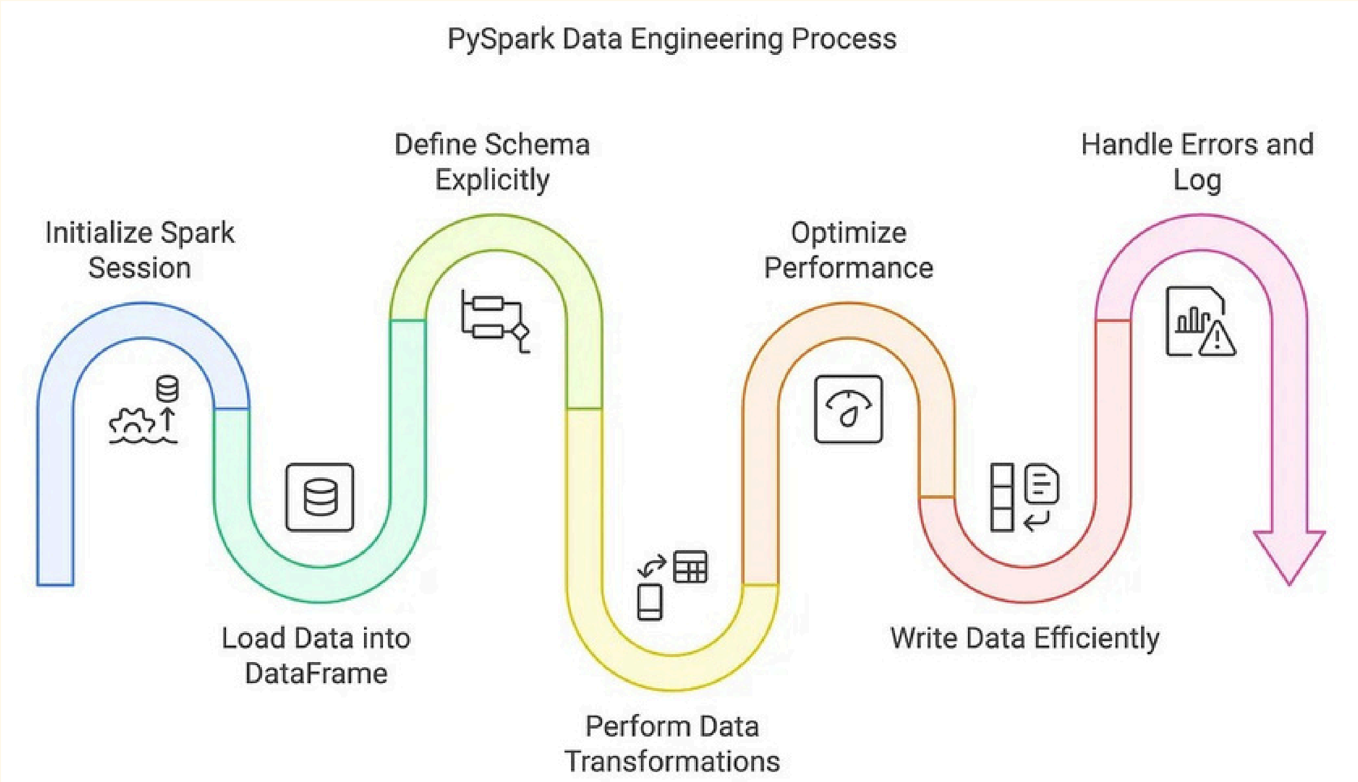
Spark — Best Practices for Data Engineer

Ganesh R

Azure Data Engineer

Introduction

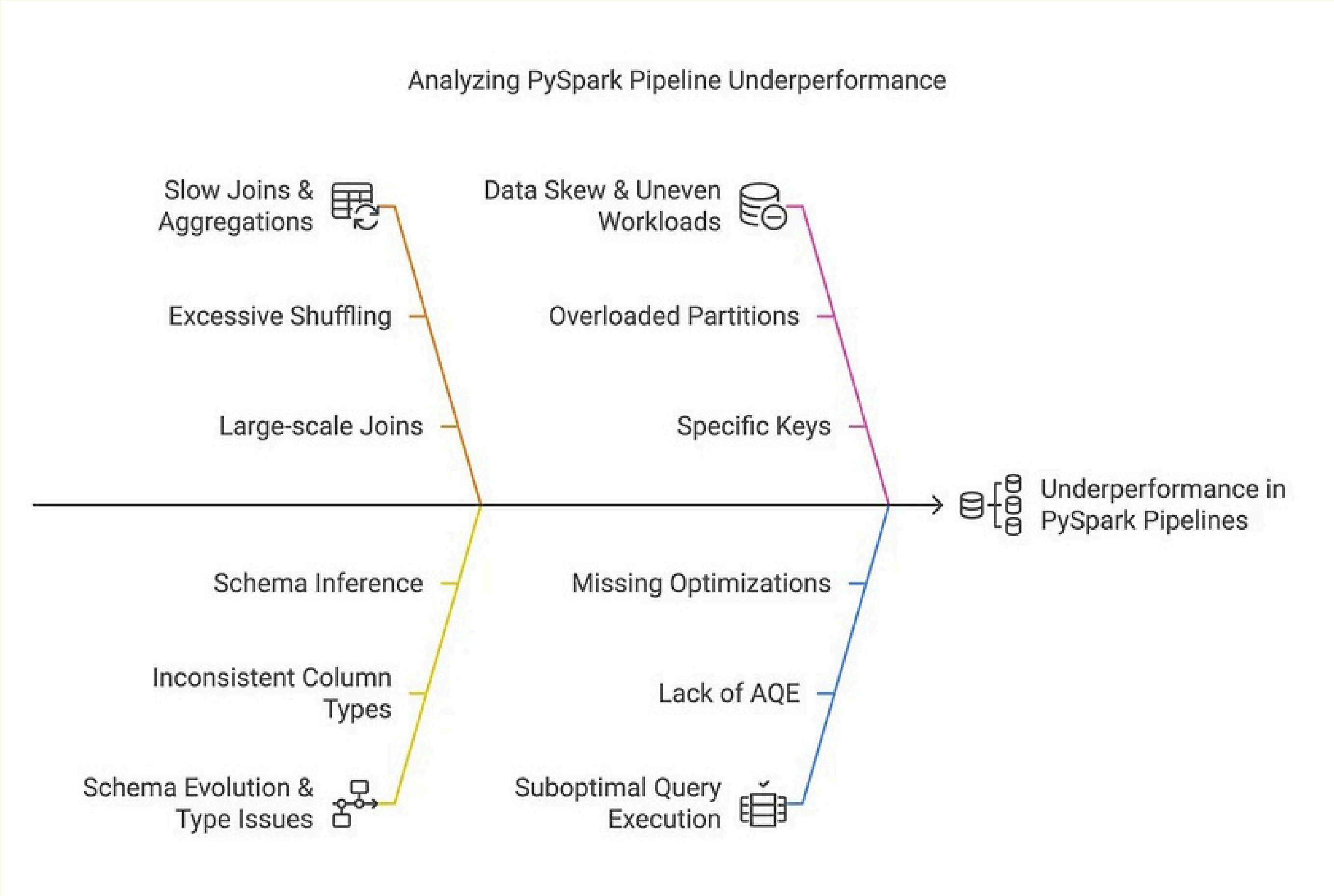
Apache Spark has become a cornerstone in big data processing, and PySpark (Spark's Python API) allows engineers to work efficiently with distributed data. However, working with large datasets at scale presents challenges, including slow transformations, inefficient joins, data skew, and poor query performance.



Spark Overloads

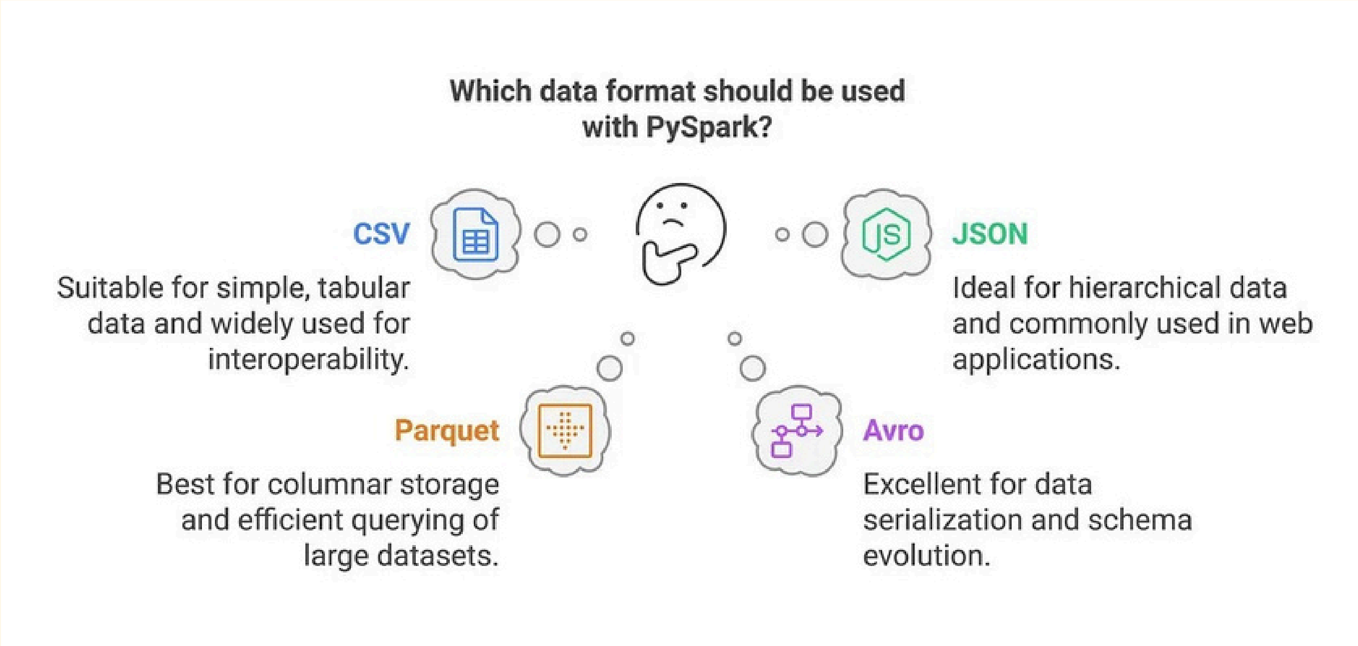
Many Data Engineers encounter these performance issues:

- Slow Joins & Aggregations → Large-scale joins causing excessive shuffling.
- Schema Evolution & Type Issues → Schema inference leading to inconsistent column types.
- Data Skew & Uneven Workloads → Certain keys overload specific partitions.
- Suboptimal Query Execution → Queries not leveraging Adaptive Query Execution (AQE) or optimizations.



Spark DataFrame

PySpark supports multiple data formats, such as CSV, JSON, Parquet, Avro, and ORC. DataFrames provide an efficient abstraction for working with structured data, and they are independent of the source or target file format. A data frame can be created from any of these formats or other sources, such as databases and in-memory collections.



The Risk of Schema Inference in Production

While using inferSchema=True is convenient, it can increase load times because Spark performs an initial scan of a subset of rows to infer the schema before performing the actual data processing. Providing the schema upfront eliminates this extra step and improves performance. However, this consideration primarily applies to formats like CSV and JSON, as Parquet, ORC, and Avro already include schema information within their metadata.

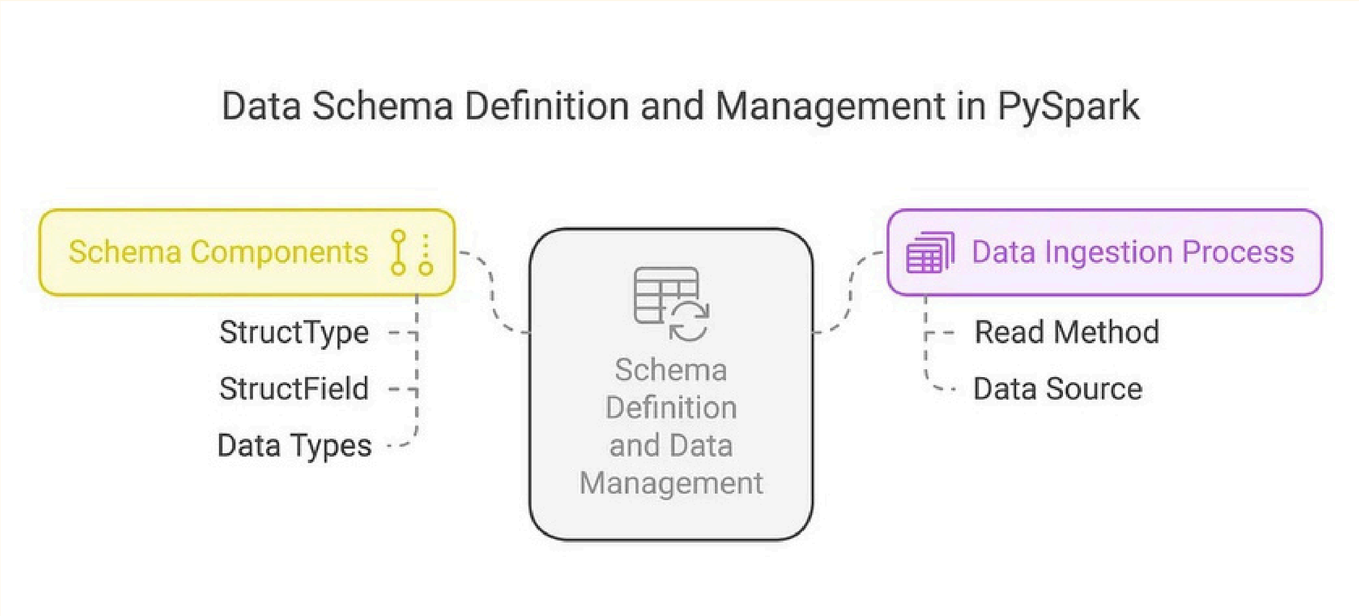
Defining Schemas Explicitly for Stability

```python
from pyspark.sql.types import StructType, StructField, IntegerType, StringType

# Define schema for stability and performance
schema = StructType([
    StructField("id", IntegerType(), True),
    StructField("name", StringType(), True),
    StructField("salary", IntegerType(), True)


df = spark.read.schema(schema).json("data.json")
```

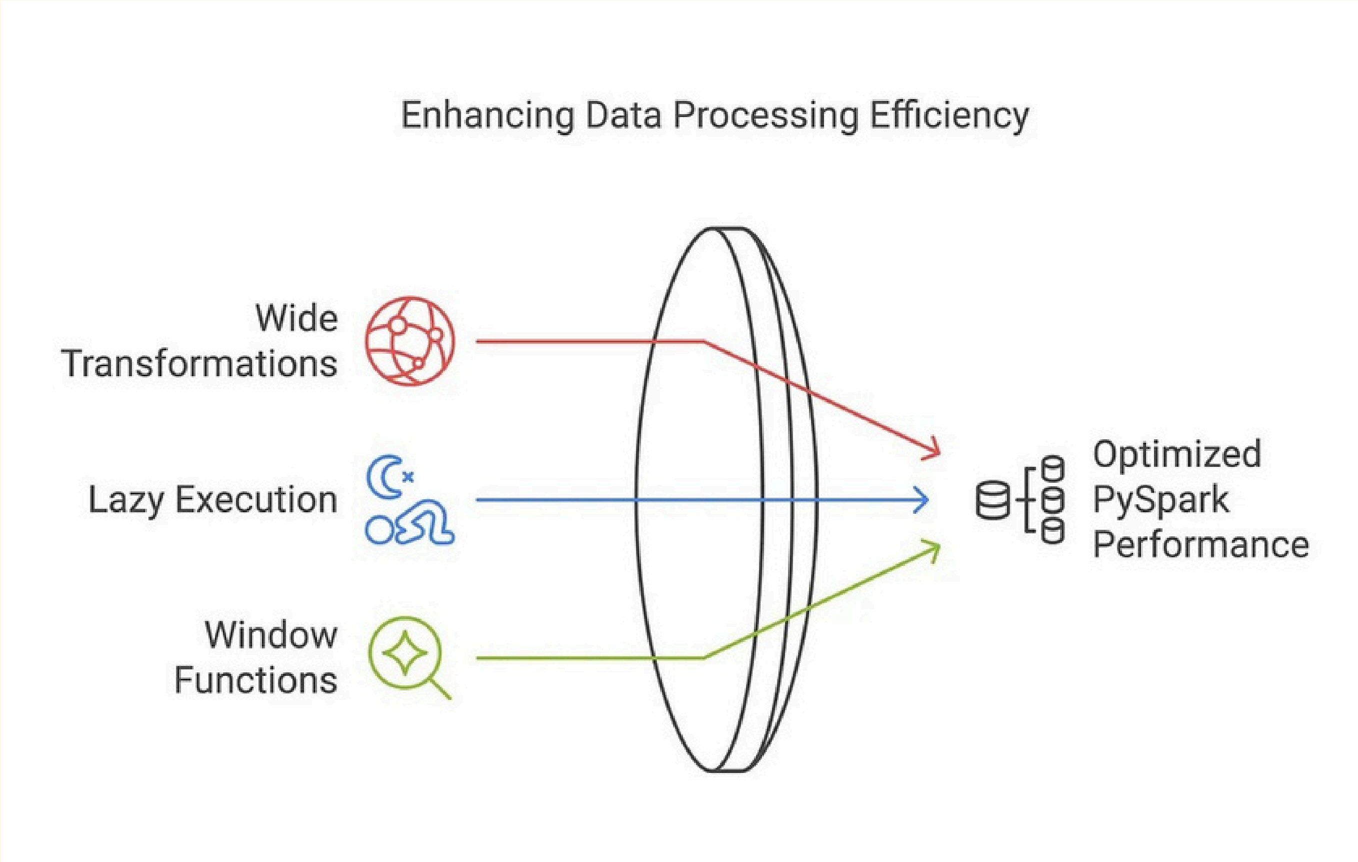
By explicitly defining schemas, Spark skips type inference, reducing unnecessary computations and ensuring data consistency.



Optimizing Data Transformations

The Problem: Why Are Some PySpark Queries Slow?

- Using wide transformations (groupBy(), join()) without optimization.
- Running transformations without understanding lazy execution.
- Not leveraging window functions for efficient aggregations.



Handling Performance Bottlenecks in PySpark

Issue 1: Data Skew in Joins

Imagine an e-commerce dataset in which 90% of transactions come from a single region. This creates data skew, overloading certain partitions while underutilizing others.

Why does Data Skew happen?

Data skew occurs when certain keys are significantly more frequent than others, leading to an imbalance in how Spark partitions and processes the data. This issue is particularly problematic in joins and aggregations, where a small subset of keys may dominate processing time.

Why Salting is Not Always the Best Solution

A common misconception is that adding a randomized salt to a join key always alleviates skew. While salting can help distribute data more evenly, it requires careful application. Specifically, one of the datasets must contain duplicate records for all possible salt values to maintain correctness. Otherwise, the join logic will break, leading to incorrect results.

Better Strategies for Handling Data Skew

Instead of relying solely on salting, consider the following alternative strategies:

1. Refine Partitioning Strategy — If possible, repartition your data using repartition() or hash-partitioning techniques to distribute load more evenly.

2. Leverage Adaptive Query Execution (AQE) — Spark's AQE can dynamically optimize shuffle partitions at runtime to balance skewed workloads:

```
spark.conf.set("spark.sql.adaptive.enabled", True)
```

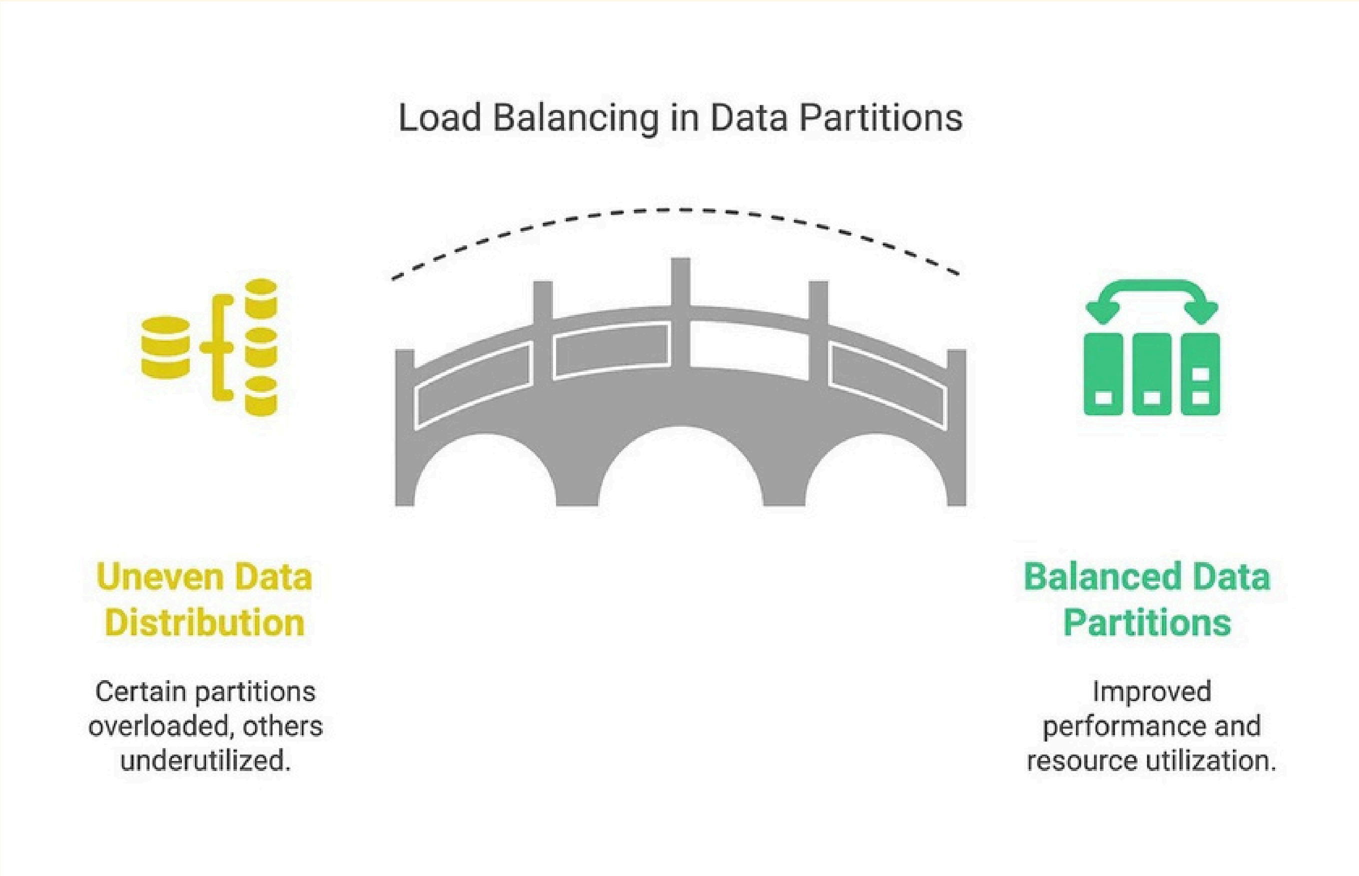
3. Skew Join Optimization — Use Spark's built-in skew join handling by enabling:

```
spark.conf.set("spark.sql.adaptive.skewJoin.enabled", True)
```

4. Broadcast Joins for Small Tables — If one table is small, use broadcasting to avoid excessive shuffling:

```
from pyspark.sql.functions import
broadcast df_joined = df.join(broadcast(small_df), "key")
```

By prioritizing partitioning improvements and AQE, you can achieve better performance without introducing potential correctness issues from improper salting.



Issue 2: Slow Joins Due to Excessive Shuffling

Large table joins lead to expensive shuffle operations, which slow down query performance.

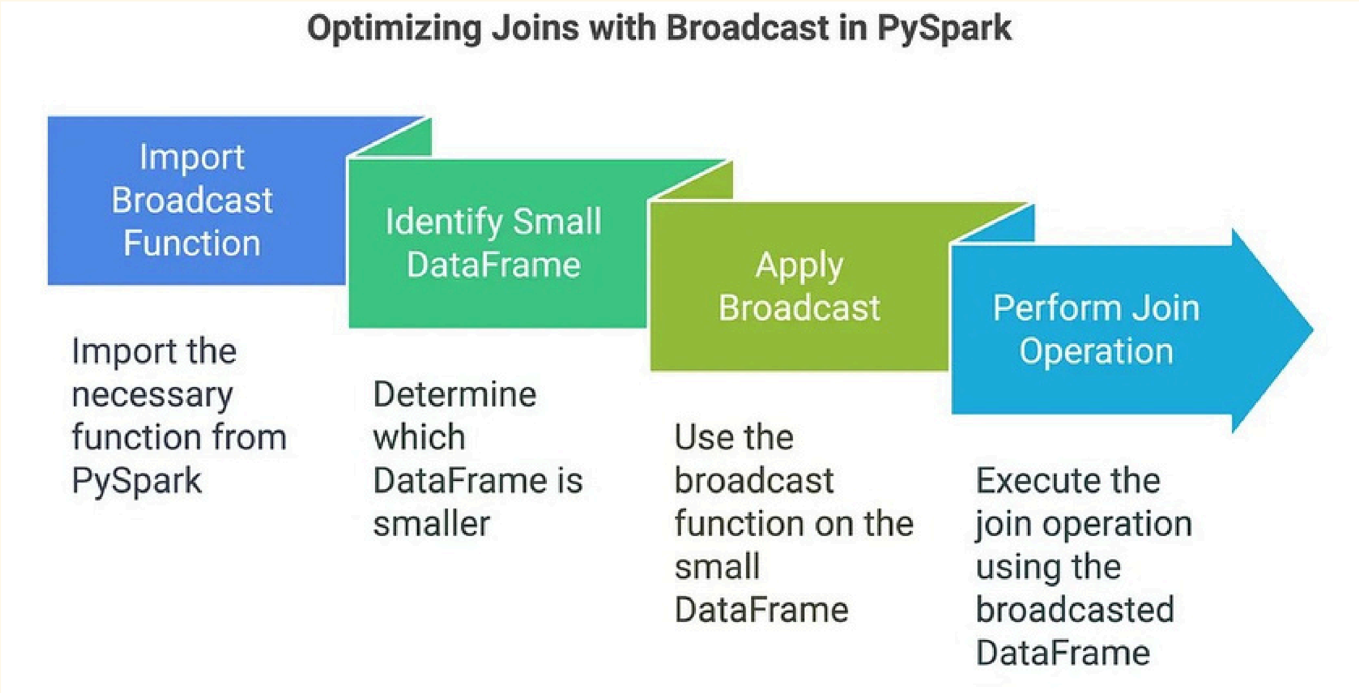
Solution:

Use Broadcast Joins for Small Tables

```
from pyspark.sql.functions import broadcast

# Broadcast small_df to optimize join
df_joined = df.join(broadcast(small_df), "key")
```

Broadcasting eliminates unnecessary shuffling by distributing the smaller data frame to all nodes.



Adaptive Query Execution (AQE): Let Spark Optimize Queries for You

Issue 3: Static Query Execution Plans

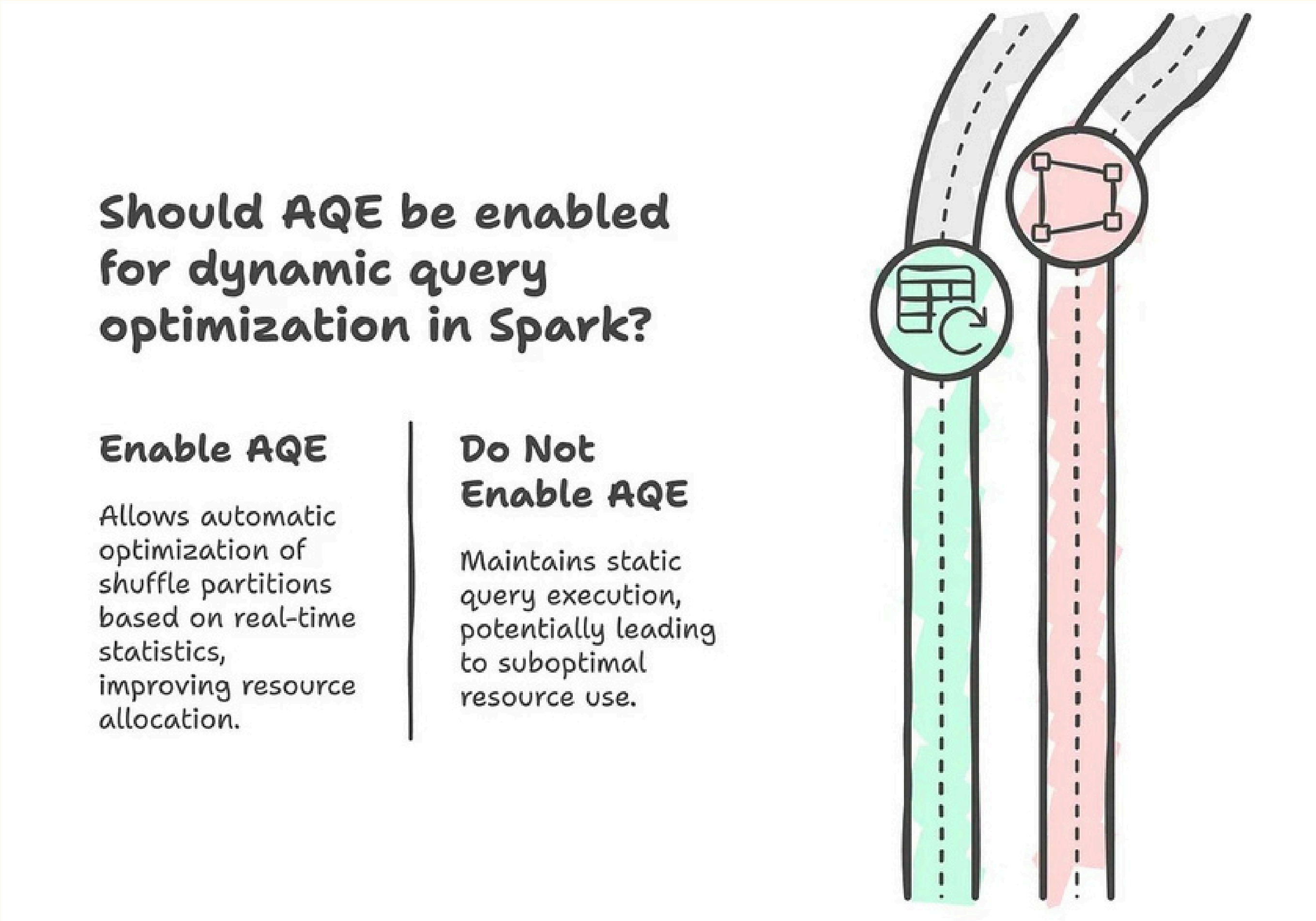
Without optimizations, Spark executes queries statically, often leading to suboptimal resource allocation.

Solution:

Enable AQE for Dynamic Query Optimization

```
spark.conf.set("spark.sql.adaptive.enabled", True)
```

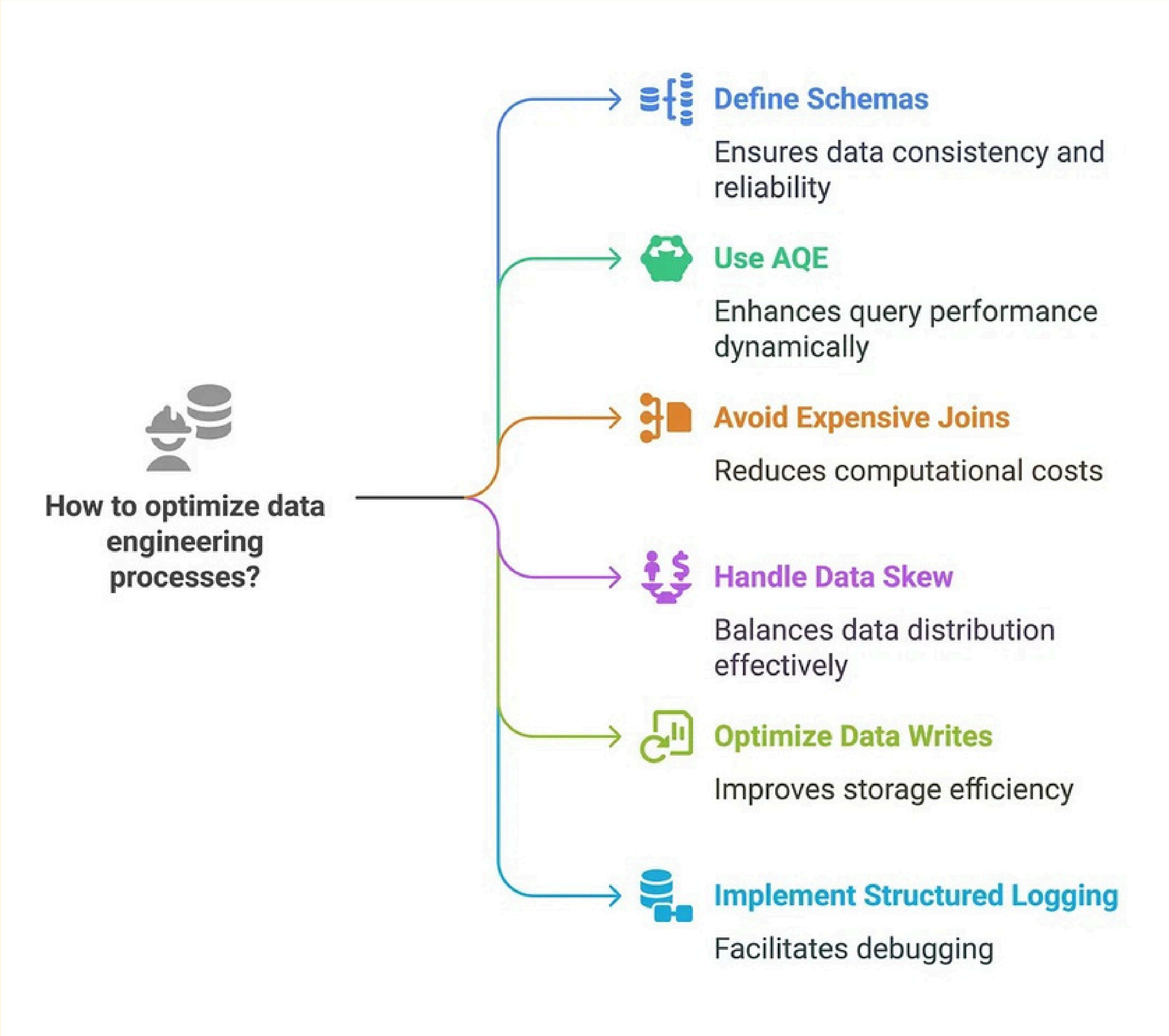
AQE automatically optimizes shuffle partitions and coalesces them based on real-time execution statistics.



Key Takeaways:

What Every Data Engineer Should Know

- ✅ Define schemas explicitly instead of relying on inference.
- ✅ Use AQE to dynamically optimize queries at runtime.
- ✅ Avoid expensive shuffle joins by using broadcast joins where possible.
- ✅ Handle data skew by using salting techniques for large datasets.
- ✅ Optimize data writes by using Parquet or Delta formats.
- ✅ Implement structured logging for debugging large-scale Spark jobs.


How to optimize data engineering processes?
Define Schemas
Ensures data consistency and reliability
Use AQE
Enhances query performance dynamically
Avoid Expensive Joins
Reduces computational costs
Handle Data Skew
Balances data distribution effectively
Optimize Data Writes
Improves storage efficiency
Implement Structured Logging
Facilitates debugging

Follow for more content like this Azure Cloud for Data Engineering

Ganesh R

Azure Data Engineer

https://www.linkedin.com/in/rganesh0203/