**How does Spark handle Fault Tolerance?**

**1. Lineage & RDD Recovery**

* Spark **tracks transformations (lineage)** and re-computes lost partitions instead of storing intermediate results.
* If an executor fails, its lost partitions are re-executed from the original dataset.

**2. Checkpointing**

* If an RDD is checkpointed, it is saved to **HDFS/S3** to avoid recomputation.

rdd.checkpoint() # Stores RDD data in storage

**3. Task Retry**

* If a task fails due to a node failure, Spark **retries it on another executor**.
* We can control this using this function.

spark.yarn.maxAppAttempts

**How does Spark perform a Shuffle?**

* A **shuffle** happens when data moves between partitions.
* **Spark performs two shuffle operations**:
  1. **Shuffle Write** – Data is written to disk.
  2. **Shuffle Read** – Other executors fetch data.

🔹 **Optimizations for Shuffle:**

* **reduceByKey() is better than groupByKey()** – It avoids sending all values across the network.
* **Map-side aggregation** – Reduces shuffle size before sending data.

**What is the difference between Spark SQL, DataFrame, and RDD?**

| **Feature** | **RDD** | **DataFrame** | **Spark SQL** |
| --- | --- | --- | --- |
| **Schema** | No schema | Schema-based | SQL-based |
| **Optimization** | No optimization | Catalyst Optimizer | Catalyst + Tungsten |
| **Performance** | Slower | Faster | Fastest |
| **Interoperability** | Supports only Scala/Python/Java | Supports multiple languages | Uses SQL queries |

**How does Catalyst Optimizer work in Spark?**

* **Catalyst** is an optimizer that improves query performance.
* It follows **four phases**:
  1. **Parse SQL Query** → Creates an **Unresolved Logical Plan**.
  2. **Bind to Dataset Schema** → Converts to a **Resolved Logical Plan**.
  3. **Apply Optimizations** → Pushdown Filters, Predicate Optimization.
  4. **Generate Physical Plan** → Executes optimized query.

**What are Broadcast Variables & Accumulators?**

**Broadcast Variables**

* Used to send a large read-only variable to all executors.
* Reduces network overhead.

**Accumulators**

* Used for distributed **counters**.
* Useful for **logging or debugging**.

| **Feature** | **Broadcast Variable** | **Accumulator** |
| --- | --- | --- |
| **Purpose** | Send read-only data to all nodes | Aggregate values across nodes |
| **Example Use** | Lookup tables | Counting errors in logs |
| **Example** |  |  |

**How does Spark optimize DataFrame operations?**

**🔹 Key points to discuss:**

1. **Catalyst Optimizer**
   * Converts SQL/DataFrame API calls into an optimized execution plan.
   * Includes **logical optimizations** (constant folding, predicate pushdown) and **physical optimizations** (join reordering, code generation).
2. **Tungsten Execution Engine**
   * Uses **bytecode generation** to optimize computation.
   * Performs **off-heap memory management** for better performance.
3. **Columnar Storage and Predicate Pushdown**
   * If using Parquet/ORC, Spark reads only the necessary columns.

**What is WholeStageCodegen in Spark?**

**WholeStageCodegen (WSCG)** is Spark's **JVM optimization** technique that:

* Converts SQL/DataFrame operations into **low-level Java bytecode**.
* Eliminates **virtual function calls & object creation**.

✅ **Checking WSCG Optimization**:

python

CopyEdit

df.explain(True) # Look for "WholeStageCodegen"

✅ **Example Execution Plan:**

bash

CopyEdit

== Physical Plan ==

\*(1) Project [name#3, age#4]

+- \*(1) Filter (isnotnull(age#4) AND (age#4 > 30))

+- FileScan parquet [name#3, age#4]

* \*(1) → Means **WholeStageCodegen is enabled**.
* **If missing?** → Spark falls back to Tungsten's **Sort-Merge Join** (slower).

**What is the difference between cache() and persist()?**

**🔹 Key differences:**

| **Feature** | **cache()** | **persist(StorageLevel)** |
| --- | --- | --- |
| **Storage** | Stores in memory (default) | Supports different storage levels |
| **Storage Levels** | MEMORY\_ONLY | MEMORY\_AND\_DISK, DISK\_ONLY, etc. |
| **Use Case** | When you need repeated access | When memory is limited, and disk storage is acceptable |



**Optimizing Joins**

1. **Broadcast Join (Best for Small Tables)**

python

CopyEdit

df.join(broadcast(df\_small), "id")

1. **Sort-Merge Join (Default)**

python

CopyEdit

df1.repartition("id").sortWithinPartitions("id")

df2.repartition("id").sortWithinPartitions("id")

df1.join(df2, "id")

1. **Shuffle Hash Join (Used when sorting isn't needed)**

**Tune Garbage Collection**

--conf spark.executor.extraJavaOptions=-XX:+UseG1GC

**How do you debug performance bottlenecks in Spark?**

**🔹 Tools to Debug**

1. **Check Spark UI (Stage Details, DAG, Shuffle Read/Write)**

bash

CopyEdit

http://localhost:4040

1. **Use explain(True) to analyze queries**

df.explain(True)

1. **Enable Event Logging**

--conf spark.eventLog.enabled=true

**Schema Evolution & Metadata Handling**

* Use **Apache Iceberg / Delta Lake** to handle schema changes.
* Maintain a **central schema registry** (e.g., Confluent Schema Registry).

✅ **Handle schema changes dynamically**:

java

CopyEdit

df.write()

.mode("append")

.option("mergeSchema", "true")

.format("delta")

.save("s3://datalake/");

**What is Backpressure in Spark Streaming, and how do you handle it?**

**🔹 Answer: Backpressure occurs when data arrives faster than Spark can process.**

✅ **How to Handle Backpressure?**

1. **Enable Adaptive Rate Control (spark.streaming.backpressure.enabled=true)**

spark.conf.set("spark.streaming.backpressure.enabled", "true");

1. **Limit Kafka Read Rate (spark.streaming.kafka.maxRatePerPartition)**

spark.conf.set("spark.streaming.kafka.maxRatePerPartition", "5000");

1. **Use Auto-Scaling with Dynamic Resource Allocation**

spark.conf.set("spark.dynamicAllocation.enabled", "true");

1. **Batch Aggregation (Reduce Per-Message Overhead)**

df.withWatermark("event\_time", "10 minutes")

.groupBy(window("event\_time", "5 minutes"), "user\_id")

.agg(count("\*").alias("total\_events"));

Important things to remember :

### **1. Spark Job**

* A **job** is triggered when you call an **action** (e.g., collect(), save(), count()).
* It represents the **entire computation** required to produce the result.
* A job is **divided into one or more stages**.

💡 **Example**: df.write.parquet() → This triggers a **Spark job** to write the DataFrame.

### 🧱 **2. Stage**

* A **stage** is a set of tasks that can be executed **without a shuffle**.
* Spark divides a job into **narrow** and **wide transformations**:
  + **Narrow transformations** (e.g., map, filter) → No shuffle → Same stage.
  + **Wide transformations** (e.g., groupBy, join) → Requires shuffle → New stage.
* Stages are either **shuffleMapStage** or **resultStage**.

💡 A job is typically split into multiple stages **based on shuffle boundaries**.

### ⚙️ **3. Task**

* A **task** is the **smallest unit of work** in Spark.
* Each task represents **a computation on a single partition**.
* A stage consists of **many tasks**—one per partition of the data.

💡 If a stage has 100 partitions, it will launch **100 tasks**.

spark.task.maxFailures:

* Number of continuous failures of any particular task before giving up on the job. The total number of failures spread across different tasks will not cause the job to fail; a particular task has to fail this number of attempts continuously. If any attempt succeeds, the failure count for the task will be reset. Should be greater than or equal to 1. Number of allowed retries = this value - 1.

**According to the recommendations which we discussed above:**

* Based on the recommendations mentioned above, Let’s assign 5 core per executors => --executor-cores = 5 (for good HDFS throughput)
* Leave 1 core per node for Hadoop/Yarn daemons => Num cores available per node = 16-1 = 15
* So, Total available of cores in cluster = 15 x 10 = 150
* Number of available executors = (total cores/num-cores-per-executor) = 150/5 = 30
* Leaving 1 executor for ApplicationManager => --num-executors = 29
* Number of executors per node = 30/10 = 3
* Memory per executor = 64GB/3 = 21GB
* Counting off heap overhead = 7% of 21GB = 3GB. So, actual --executor-memory = 21 - 3 = 18GB

**AQE** in Spark stands for **Adaptive Query Execution**.

It is a feature introduced in **Apache Spark 3.0** that dynamically optimizes query plans at runtime, based on the actual data statistics collected during query execution. Prior to AQE, Spark relied only on static query planning based on estimated statistics, which could lead to suboptimal execution due to inaccurate estimates.

### 🔧 **Key Features of AQE:**

### **Dynamic Join Strategy Selection**

* + Spark can **switch join types** (e.g., from Sort-Merge Join to Broadcast Join) during execution if it finds one side of the join is small enough to broadcast.

### **Dynamic Partition Pruning**

* + Spark can **prune unnecessary partitions** during query execution rather than at compile time, which reduces I/O.

### **Handling Skewed Joins**

* + AQE can detect **data skew** at runtime and **split skewed partitions** into smaller ones to avoid stragglers.

### **Coalescing Shuffle Partitions**

* + Based on the size of data in each shuffle partition, AQE can **dynamically reduce the number of shuffle partitions**, improving performance and reducing overhead.

### 🔍 **How to Enable AQE:**

In Spark 3.0+, it is **disabled by default**, but you can enable it:

python

CopyEdit

spark.conf.set("spark.sql.adaptive.enabled", "true")

Other related configs:

python

CopyEdit

spark.conf.set("spark.sql.adaptive.coalescePartitions.enabled", "true")

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

spark.conf.set("spark.sql.adaptive.localShuffleReader.enabled", "true")

### 📊 Example Scenario:

You expect a large dataset on both sides of a join, so you plan a **Sort-Merge Join**. But during execution, Spark finds one side is small — AQE **switches** to **Broadcast Join**, which is more efficient.

### ✅ **Benefits of AQE:**

* Better performance without manual tuning
* Reduced impact of incorrect statistics
* Automatically handles data skew and partition imbalance