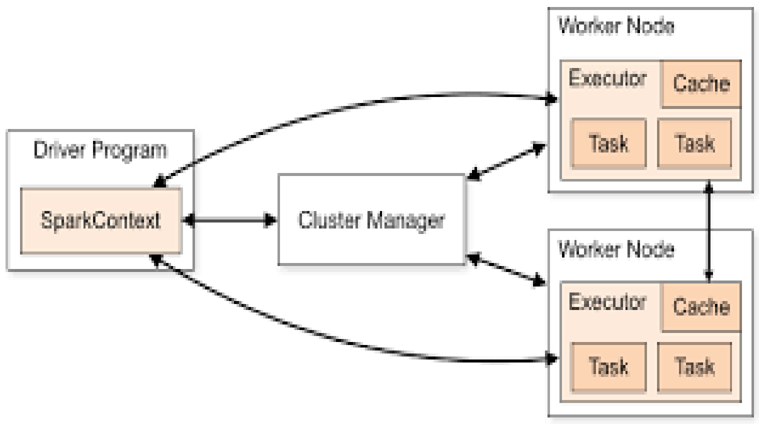
Spark Architecture



Key Features of Apache Spark

- **Unified Engine → Supports batch, streaming, SQL, machine learning, and graph processing under one framework.**
- **In-Memory Computation → Up to 100x faster than disk-based systems like Hadoop MapReduce.**
- **DAG Execution Engine → Optimized query execution with lineage tracking for fault tolerance.**
- **Scalability → Handles data from GBs to PBs across thousands of nodes.**
- **Fault Tolerance → RDD lineage + DAG recomputation + checkpointing in streaming.**
- **Advanced Optimizers → Catalyst Optimizer (query plans) and Tungsten Engine (memory & CPU optimization).**
- **Rich Integration → Works seamlessly with data lakes (ADLS, S3), warehouses (Synapse, Snowflake), and messaging systems (Kafka, Event Hub).**
- **Flexible Deployment → Runs on YARN, Kubernetes, Mesos, Standalone, and cloud services like Azure Databricks.**

Spark Architecture (High-Level View)

- **Driver Program**
  - Runs the **main application**.
  - Creates **SparkSession** (entry point).
  - Builds **Logical Plan → Optimized Plan → Physical Plan**.
  - Responsible for **job scheduling and metadata management**.

- **Cluster Manager**
  - Allocates resources (CPU, memory) for executors.
  - Pluggable managers: **YARN, Kubernetes, Standalone, Mesos**.
  - In Azure, **Databricks manages clusters automatically**.

- **Executors**
  - JVM processes on worker nodes.
  - Execute **tasks** assigned by driver.
  - Provide **in-memory caching** for intermediate data.
  - Long-lived for the duration of the Spark application.

- **Tasks**
  - Smallest unit of execution (one per data partition).
  - Run inside executors in parallel.

- **DAG Scheduler**
  - Splits job into **stages and tasks**.
  - Defines shuffle boundaries.

- **Task Scheduler**
  - Submits tasks to cluster manager.
  - Optimizes for **data locality**.

Real-Time Example: Healthcare – IoT Patient Monitoring

Hospitals collect continuous patient vitals (heart rate, blood pressure, SpO_2) from IoT medical devices.

- Doctors need **real-time alerts** when abnormal readings occur.
- All vitals should be **stored in Delta Lake** for future analysis.

⚡ Spark Execution Flow in Databricks

1. **Job Submission**
   - A **PySpark Structured Streaming job** is deployed in Databricks.
   - Streaming source = **JSON files arriving in Azure Data Lake Storage (ADLS)** every few seconds (simulating IoT feed).

2. **Driver Node**
   - Initializes `SparkSession`.
   - Reads streaming files from ADLS into a **DataFrame**.
   - Defines transformations:
     - `filter()` abnormal vitals (e.g., HR > 150).
     - `withColumn()` to calculate risk score.
     - `writeStream` to output alerts + store all records in Delta Lake.
   - Converts logical plan → optimized physical plan → DAG of stages.

3. **DAG Scheduler**
   - Splits job into **stages & tasks**. Example:
     - Stage 1 → Read JSON from ADLS.
     - Stage 2 → Apply filters & transformations.
     - Stage 3 → Write to Delta tables.

4. **Cluster Manager (Databricks Runtime)**
   - Allocates executors across worker nodes.
   - Auto-scales if more IoT data files arrive.

5. **Executors (Worker Nodes)**
   - Each executor processes partitions of patient data.
   - Executes transformations in memory, writes alerts and records to storage.
   - Sends task status to driver.

6. **Results**
   - **Critical alerts** → Written to an **alerts Delta table** (queried by doctors).
   - **Full patient history** → Written to a **bronze/silver Delta Lake table** for analytics and ML models.

Simplified Flow Chart

```
IoT Devices → ADLS (Streaming JSON Files)
        |
        ▼
Databricks Driver
   - SparkSession init
   - DAG Scheduler builds stages
        |
        ▼
Cluster Manager
   - Allocates Executors
        |
        ▼
Executors (Worker Nodes)
   - Parallel processing
   - Transform/filter vitals
   - Write results
        |
        ▼
Delta Lake (Storage)
   - Alerts Table (Critical events)
   - History Table (All vitals)
```

Spark Execution Flow

1. **Job Submission**
    - User submits application via `spark-submit`, notebook (e.g., Azure Databricks), or API call.
    - The **Driver Program** starts and initializes a **SparkSession**.
2. **Driver Builds Logical Plan**
    - User code (e.g., DataFrame transformations like `filter`, `join`) is converted into a **Logical Plan**.
    - Catalyst Optimizer refines it into an **Optimized Logical Plan**.
3. **Physical Plan & DAG Creation**
    - Spark generates a **Physical Plan** → breaks job into **stages** (based on shuffle boundaries).
    - Each stage is further split into **tasks** (per partition).
4. **Cluster Manager Allocates Resources**
    - YARN/Kubernetes/Databricks assigns **Executors** (JVMs on worker nodes).
5. **Task Execution on Executors**
    - Executors run tasks in parallel.
    - Intermediate results cached in **memory/disk**.
    - Executors periodically send **heartbeat & task status** to the driver.
6. **Result Collection / Storage**
    - Results are sent back to the Driver if small.
    - Large outputs are written to external storage (ADLS, Delta Lake, Synapse, Kafka, etc.).

Real Use Case Examples

- **Banking** → SCD Type-2 history tables in Delta Lake for KYC compliance.
- **Retail** → Customer 360 view combining POS + e-commerce data for personalization.
- **Telecom** → Real-time churn prediction using Spark Streaming + MLlib.
- **FMCG** → Supply chain optimization with IoT streaming data via Event Hub → Spark → Synapse.

Q.1 How many stages are created while performing an action in Databricks?

Ans. The number of stages created when performing an action in Databricks (powered by Apache Spark) is not fixed — it depends on the transformations applied before the action.

Key Points:

- Stages are created during actions like collect(), count(), save(), etc.

- Spark breaks your job into stages based on shuffle boundaries.

- Each wide transformation (e.g., groupBy, join, distinct) introduces a new stage.

- Narrow transformations (e.g., map, filter) are grouped into a single stage.

Example:

python

```
df = spark.read.csv("data.csv")

df_filtered = df.filter(df["value"] > 100)

df_grouped = df_filtered.groupBy("category").count()

df_grouped.collect()
```

This may result in:

- Stage 1: Read + filter (narrow transformations)

- Stage 2: groupBy (wide transformation → shuffle)

- Stage 3: collect (final action)

Answer: The number of stages is dynamic, depending on your transformation pipeline. You can view the stages in the Spark UI under the "Jobs" tab in Databricks.

Q.2 How many stages are created in a wide and narrow transformation in Databricks

Databricks (Apache Spark), the number of stages created depends on whether the transformations are narrow or wide, and how they are arranged in your pipeline.

Key Concept:

- Stages are created at shuffle boundaries, not directly by narrow or wide transformations themselves.

Narrow Transformations

- Examples: map, filter, select, withColumn, etc.

- Data flows from one partition to another without shuffling.

- Do not create new stages on their own — they're grouped into a single stage.

➡ Result: Multiple narrow transformations → 1 stage.

Wide Transformations

- Examples: groupBy, join, distinct, reduceByKey, etc.

- Require data shuffling across the network.

- Each wide transformation creates a new stage because of the shuffle.

➡ Result: Each wide transformation → new stage.

Example Workflow:

python

```
df = spark.read.csv("data.csv")

df1 = df.filter("age > 30")         # Narrow

df2 = df1.groupBy("city").count()   # Wide → Shuffle → Stage boundary

df3 = df2.filter("count > 10")      # Narrow

df3.collect()                       # Action
```

Stages Created:

1. Stage 1: Read + filter (narrow)

2. Stage 2: groupBy (wide → shuffle)

3. Stage 3: filter + collect (narrow + action)

3.How many default number of partitions are created when you're shuffling and doing a wide transformation

Ans - 200 partitions

Databricks (Apache Spark), when a *wide transformation (like groupBy, join, distinct, etc.) causes a shuffle, the default number of partitions created during the shuffle is:

200 partitions by default-This is controlled by the Spark configuration:

Notes:

- You can change this value based on data size or performance needs:

python

spark.conf.set("spark.sql.shuffle.partitions", 100)

- Having too many partitions can lead to overhead (too many small tasks).

- Too few partitions may lead to data skew and underutilization of resources.

Tip:

Always tune spark.sql.shuffle.partitions based on:

- Size of the data being shuffled

- Number of cores in the cluster

- Desired parallelism

spark.sql.shuffle.partitions = 200

q.4 A Spark job is running very slowly. How would you debug and optimize it?

Debugging a Slow Spark Job

1. **Check the Spark UI (Stages & DAG visualization)**
   - Look at **stage execution times** → identify straggler stages.
   - Look at **task distribution** → skew, uneven partition sizes, or long-running tasks.
   - Look at **shuffle read/write sizes** → heavy shuffles usually indicate joins, groupBy, or repartition problems.

2. **Understand the DAG**
   - Is the job overcomplicated? Too many unnecessary transformations?
   - Are there wide transformations (e.g., joins, groupBy, distinct) that trigger heavy shuffles?

3. **Look for Skew**
   - In joins/aggregations → some keys may be huge compared to others (data skew).
   - Check the Spark UI → tasks processing **GBs vs MBs** of data are red flags.

4. **Memory & GC (Garbage Collection) issues**
   - Executors spending a lot of time in GC? → Too high memory fraction for shuffle or caching.
   - Use spark.executor.memory, spark.memory.fraction, and caching only when reused.

5. **I/O bottlenecks**
   - Input format → Are you reading from inefficient storage (small files in S3/ADLS, no partition pruning in Parquet/Delta)?
   - Output format → Too many small output files, no coalesce/repartition before writing.

⚡ Optimization Techniques

1. **Data Partitioning & Parallelism**
    - Use `repartition()` or `coalesce()` wisely to balance tasks.
    - Match partitions with cluster cores (avoid under-parallelization or excessive partitions).
    - Enable **predicate pushdown** in Parquet/Delta to avoid scanning full datasets.

2. **Reduce Shuffles**
    - Use **broadcast joins** (`broadcast(df2)`) when one side is small (<100MB).
    - Use **bucketing** or **partitioning** on frequently joined/aggregated columns.
    - Avoid `groupByKey`, prefer `reduceByKey` or `mapPartitions`.

3. **Handle Skew**
    - Use **salting** (adding random keys to distribute skewed data).
    - Use **adaptive query execution (AQE)** in Spark 3+ (`spark.sql.adaptive.enabled=true`).

4. **Caching & Persistence**
    - Cache only if reused multiple times.
    - Unpersist once not needed.
    - Prefer `persist(StorageLevel.MEMORY_AND_DISK)` for large data.

5. **Efficient Data Formats**
    - Use **Parquet/Delta** with **compression** (snappy/zstd).
    - Use **partition pruning** (filter on partitioned columns).

6. **Cluster & Config Tuning**
    - Right-size executors:

    - spark.executor.cores = 4–5 (avoid too many per executor).
    - spark.executor.memory tuned for dataset size.
  - Enable **dynamic allocation** if workloads vary.
  - Use **speculative execution** (spark.speculation=true) to handle stragglers.

Example Debug Flow (Real-Time Case)

Imagine you're running a **claims processing pipeline in healthcare**:

- Input: Raw claim files (JSON) → Delta
- Processing: Joins with patient master data + aggregation by provider
- Issue: Job taking 2 hours instead of 20 minutes

Debugging steps:

- Spark UI → one stage taking 90 mins → `groupBy(provider_id)` aggregation.
- Found **data skew**: One provider had 60% of claims.
- Fix: Added **salting technique** + **AQE enabled**.
- Reduced runtime → 25 minutes.

Q.5 Your Databricks job fails due to "out of memory" error. How do you fix it?

Why Spark Jobs Fail with OOM in Databricks

1. **Skewed data** → some tasks handle GBs of data while others handle MBs.
2. **Improper partitioning** → too few (large) partitions = heavy tasks.
3. **Wide transformations** (e.g., big joins, groupBy, distinct) → shuffles blow up memory usage.
4. **Too much caching/persistence** → cached data filling executor memory.
5. **Executor misconfiguration** → too many cores per executor, not enough memory.

Fixes for OOM Errors in Databricks

1. Tackle Data Skew

- Check the **Spark UI** → if one task is much larger than others → skew.
- Solutions:
  - Use **salting** (add a random key to distribute skewed keys).
  - Use **Spark AQE** (spark.sql.adaptive.enabled = true) → Databricks supports it.
  - Consider **broadcast join** if one table is small.

---

2. Optimize Partitioning

- Too few partitions → big partitions = OOM.
- Fix: Increase partitions
- df = df.repartition(200)   # balance load
- Too many partitions → overhead. Use coalesce() before writing output.
- In Databricks, you can monitor partition size with **Data Skew visualization** in UI.

---

3. Reduce Shuffle Memory Usage

- Avoid groupByKey, use reduceByKey or mapPartitions.
- Use **join hints**:
- from pyspark.sql.functions import broadcast
- df = large_df.join(broadcast(small_df), "id")
- Enable **AQE** for optimized shuffle partition sizes.

---

4. Optimize Caching & Persistence

- Only cache() if reused multiple times.
- Use persist(StorageLevel.MEMORY_AND_DISK) instead of memory-only.
- Drop cache after use:
- df.unpersist()

---

5. Tune Cluster & Spark Config

- Right-size executors:
  - Too many cores per executor = memory pressure.
  - Try spark.executor.cores=4 (not max).

- Increase memory: scale cluster vertically in Databricks (worker type → larger RAM).
- Enable speculative execution for stragglers:
- spark.conf.set("spark.speculation", "true")

---

6. Input/Output Data Optimizations

- Use **Parquet/Delta** instead of JSON/CSV.
- Use **partition pruning** (filter on partitioned columns).
- Compact small files before processing (OPTIMIZE in Delta).

q.6 How do you configure cluster types (Job cluster vs All-purpose cluster) in Databricks for cost efficiency?

Cluster Types in Databricks

1. Job Cluster

- Created **per job run** (via Jobs UI, API, or notebook task).
- **Ephemeral** → spins up when the job starts, shuts down when the job ends.
- Best for **production pipelines, scheduled ETL jobs, or one-time batch workloads**.
- Pros:
  - **Cost efficient**: No idle time → you only pay while job runs.
  - Automatically scales based on workload (if autoscaling enabled).
- Cons:
  - Cold start latency (cluster spin-up time ~2–5 mins).
  - Not great for ad-hoc queries.

---

2. All-purpose Cluster

- **Interactive & shared** → notebooks, dashboards, ad-hoc analysis.
- Can be attached to multiple notebooks and users.
- Best for **exploratory analysis, development, debugging**.
- Pros:
  - Fast iteration (cluster already running).
  - Easy collaboration.
- Cons:
  - **Expensive if left idle** (charges continue while cluster is running).

  - Risk of uncontrolled costs if multiple users run heavy jobs.

⚡ Cost Efficiency Best Practices

For Job Clusters (Production/ETL)

- Use **job clusters** for scheduled workloads — they auto-terminate after completion.
- Enable **autoscaling** (min/max workers).
- Example config:
- Worker type: Standard_DS3_v2 (cost-effective, balanced)
- Min workers: 2
- Max workers: 10
- Auto-termination: Enabled
- Runtime: Latest DBR with Photon (for faster queries, lower cost)

For All-purpose Clusters (Exploration/Dev)

- **Keep small**: fewer workers, smaller node type.
- Enable **auto-termination after X mins idle** (e.g., 15–30 mins).
- Use **cluster pools** → reduce startup time while reusing VMs, saves cost.
- Share across team (but monitor quotas to avoid "cluster hogging").

Decision Guide

- **Production ETL/Batch Jobs** → Job Cluster
- **Exploratory analysis, ML model dev, debugging** → All-purpose Cluster
- **High concurrency BI workloads** → SQL Warehouse (not clusters)