Resume Summary and Skills



Great—here's a concise, technically-focused "Tell me about yourself" answer tailored for a project-level technical interview:

I'm Hari Kishore, a Data Engineer with 5.6 years of experience, including over 3.6 years in core data engineering. My primary focus is on building and optimizing scalable ETL pipelines using tools like **Apache Spark (PySpark)**, **AWS Glue**, **Athena**, **EMR**, and **Airflow**.

Currently, I'm working at Deloitte South Asia on the **IDFC FIRST Bank** project, where I've led efforts in optimizing Spark jobs—achieving up to 95% cost reduction—and automated campaign data pipelines across multiple AWS services. I've also built a **Hyper-Personalized Engine** that integrates with SMS, email, WhatsApp, and push notifications.

In my previous role at Cognizant, I developed Python-based automation tools and worked on a predictive model for batch job failures, which earned a **bronze medal** in a company-wide hackathon.

I'm Databricks certified, strong in SQL tuning, and passionate about designing **cost- effective, high-performance data workflows** in the cloud.

Would you like a version tailored for a specific project you're interviewing for?



Great technical question! Here's a clear and interview-ready explanation of **Apache Spark anatomy** and **how Spark works**:

Spark Anatomy (Architecture)

Apache Spark is built on a **master-slave architecture** and consists of the following main components:

1. **Driver Program**:

- Runs on the driver node.
- Contains your main function and SparkContext.
- Responsible for job coordination, creating the DAG (Directed Acyclic Graph),
 and breaking it into stages and tasks.

2. Cluster Manager:

- Allocates resources across the cluster (e.g., YARN, Kubernetes, Standalone, or Mesos).
- Spark integrates with these to manage executors on worker nodes.

3. Workers (Executors):

- These are processes running on worker nodes.
- Executors:
 - Run individual tasks.
 - Store data in memory/disk across stages.
 - Report back to the driver.

4. Tasks:

- The smallest unit of execution.
- Each task is a unit of work sent to an executor by the driver.

How Spark Works (Execution Flow)

1. Driver Starts:

- The driver program initializes the SparkSession (which internally creates the SparkContext).
- Connects to the cluster manager to request resources.

2. Job Submission:

When an action is triggered (like `collect()`, `count()`, `save()`), Spark builds a logical DAG of transformations.

3. DAG Scheduler:

- Breaks the DAG into stages (based on wide vs narrow transformations).
- Narrow transformation (e.g., `map`) ⇒ same partition.
- Wide transformation (e.g., `groupBy`, `join`) ⇒ shuffle between partitions.

4. Task Scheduler:

Converts stages into tasks and distributes them to executors.

5. Execution:

 Executors process the tasks, cache data (if needed), and return results to the driver.

6. Result Collection:

Driver aggregates the results and completes the job.

Example (Real-World Analogy):

Think of the driver as a **project manager**:

- It splits the job (DAG) into smaller work units (stages & tasks).
- Workers (executors) are like team members executing the tasks.
- The cluster manager is like HR or admin, allocating resources (who works on what).

Would you like a diagram or simple code example to go with this explanation?



The **Catalyst Optimizer** is a key component of **Apache Spark's SQL engine** (Spark SQL). It's responsible for **automatically optimizing** query plans to improve performance.

What is Catalyst Optimizer?

Catalyst is **Spark SQL's query optimization framework**. It uses a combination of **rules-based** and **cost-based** optimization techniques to transform logical query plans into efficient physical execution plans.

Why is it Important?

Without Catalyst, Spark would execute your SQL or DataFrame operations **exactly as** written, which could be slow or inefficient. Catalyst makes intelligent decisions to:

- Minimize data movement
- Reduce computation
- Push operations closer to the data

Catalyst Optimization Process (4 Phases)

1. Analysis:

- Parses your SQL/DataFrame code.
- Checks syntax and resolves attributes using a catalog (e.g., table and column names).

2. Logical Optimization:

- Applies rule-based optimizations like:
 - Predicate pushdown
 - Constant folding
 - Null filtering
 - Removing unnecessary filters or projections

3. Physical Planning:

- Generates multiple physical execution plans using different strategies (e.g., hash join vs broadcast join).
- Assigns a cost to each plan.

- 4. Cost-Based Optimization (CBO) (optional but powerful):
 - Chooses the most efficient plan using table statistics (e.g., row counts, data size).

🚀 Example Optimizations Catalyst Can Do

- Predicate Pushdown: Moves filters as close to the data source as possible.
- Column Pruning: Reads only necessary columns instead of the whole dataset.
- **Join Reordering**: Picks the most efficient join order based on data size.
- Broadcast Joins: Broadcasts a small table to all workers instead of shuffling both.

Real-World Analogy

Think of Catalyst as a **smart compiler** for SQL/DataFrame code. You write high-level instructions, and Catalyst figures out the **fastest** way to run them behind the scenes.

Would you like an example showing Catalyst optimization in action using DataFrame code?



Great question! Understanding **jobs**, **stages**, and **tasks** is crucial for working effectively with **Apache Spark**. Here's a clear breakdown:

🔧 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.

" **Fixample**: `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.
 - " P 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.

" P If a stage has 100 partitions, it will launch 100 tasks."

Flow Summary

```
Action (e.g., df.count())

→ Job

→ Stages (e.g., Stage 0, Stage 1)

→ Tasks (e.g., Task 0 to Task N per stage)
```

X Example

```
df = spark.read.parquet("s3://my-data")
result = df.filter("age > 30").groupBy("city").count()
result.show()
```

- Job: Triggered by `.show()`.
- **Stage 1**: `read` + `filter` (narrow transformation, no shuffle).
- Stage 2: `groupBy` + `count` (wide transformation → shuffle).
- Each stage executes tasks on partitions.

Spark UI View

When you run a job, you can view:

Number of jobs

- Stages within each job
- Number of tasks per stage
- Task duration, data shuffles, failures, etc.

Would you like a real code example with `.explain()` to see how stages are formed?



Great! Understanding **narrow vs. wide transformations** is essential to mastering **Apache Spark performance tuning**, especially for partitioning and shuffling.

Narrow Transformations

- Definition: Each partition of the parent RDD/DataFrame is used by only one partition of the child.
- No shuffle happens between executors.
- These are faster and more efficient.

Examples:

- `map()`
- `filter()`
- `flatMap()`
- `union()` (when no re-partitioning is involved)

Why "narrow"?

Think of it like a **straight path** from input to output—data flows **linearly**.

Wide Transformations

- Definition: Multiple child partitions depend on multiple parent partitions.
- Requires data shuffling across the network.
- These are expensive and may cause stage boundaries.

X Examples:

- groupByKey()
- `reduceByKey()`
- `join()`
- `distinct()`
- repartition()

Why "wide"?

These involve data redistribution, so Spark must move data across the cluster, introducing network I/O and latency.

■ Comparison Table

| Feature | Narrow Transformation | Wide Transformation |
|---------------|-----------------------|---------------------|
| Data movement | No shuffle | Involves shuffle |

| Feature | Narrow Transformation | Wide Transformation |
|-----------------|-----------------------|---------------------|
| Stage split | Same stage | New stage starts |
| Performance | Fast | Slower (due to I/O) |
| Examples | `map`, `filter` | `groupBy`, `join` |
| Dependency type | One-to-one | Many-to-many |

★ Visual Analogy

- Narrow: One worker takes one file and processes it directly.
- Wide: Multiple workers send chunks of their data to other workers for processing.

Real-World Example

```
# Narrow transformation
df = df.filter(df.age > 25)

# Wide transformation
df = df.groupBy("department").count()
```

- `.filter()` is narrow (no shuffle).
- `.groupBy()` is **wide** (triggers a shuffle → new stage).

Would you like a visual diagram or `.explain()` output to see this in action?



Excellent question — this is a common topic in Spark interviews, especially when it comes to **performance tuning**. Let's break down the difference between `groupByKey` and `reduceByKey` in Apache Spark (RDD API):

`groupByKey()`

What it does:

Groups all values with the same key into a **list**.

Mechanism:

All values for a key are **shuffled** across the network to a single reducer, **then grouped**.

- Performance:
 - Less efficient because it moves all values across the network before aggregation.
- Use case:

Use only when you need to access **all values per key** (e.g., computing median, sorting values per key).

Example:

```
rdd = sc.parallelize([("a", 1), ("a", 2), ("b", 1)])
rdd.groupByKey().mapValues(list).collect()
# Output: [('a', [1, 2]), ('b', [1])]
```

reduceByKey()`

• What it does:

Combines values **locally** on each partition **before** shuffling data.

Mechanism:

Performs **local aggregation** first \rightarrow **less data** is shuffled \rightarrow then merges final results.

Performance:

✓ More efficient and preferred for large datasets when doing aggregations.

Use case:

Use when you're doing operations like `sum`, `count`, `max`, etc.

Example:

```
rdd = sc.parallelize([("a", 1), ("a", 2), ("b", 1)])
rdd.reduceByKey(lambda x, y: x + y).collect()
# Output: [('a', 3), ('b', 1)]
```

■ Comparison Table

| Feature | `groupByKey()` | `reduceByKey()` |
|---------------|------------------------------------|------------------------------------|
| Data shuffled | All values | Only aggregated results |
| Local combine | X No | ▼ Yes |
| Performance | Slower, more network I/O | ▼ Faster, less data shuffle |
| Use case | When all values are needed per key | When performing aggregation |

Interview Tip:

""In Spark, prefer `reduceByKey` over `groupByKey` for aggregation tasks to avoid unnecessary data shuffling and improve performance.""

Would you like a PySpark version of this or a DataFrame equivalent comparison (`groupBy().agg()`)?



Great question — this comparison is about understanding the **difference between** 'map()' and 'filter()' in **Apache Spark** (RDD/DataFrame context). These are both narrow transformations, but they serve different purposes.

- `map()`
- Purpose: Transforms each element of the RDD/DataFrame individually.
- Returns: A new RDD/DataFrame with the same number of elements, but transformed.
- **Use case**: When you want to **change** the contents (e.g., multiply, format, parse, etc.).

Example:

```
rdd = sc.parallelize([1, 2, 3])
rdd.map(lambda x: x * 2).collect()
# Output: [2, 4, 6]
```

- `filter()`
- Purpose: Keeps only the elements that meet a condition.
- Returns: A new RDD/DataFrame with fewer elements (or the same, if all match).
- Use case: When you want to exclude or include data based on a condition.

Example:

```
rdd = sc.parallelize([1, 2, 3, 4])
rdd.filter(lambda x: x % 2 == 0).collect()
# Output: [2, 4]
```

Combined Use Case

You can use `filter()` and then `map()` together, often seen in data preprocessing:

```
rdd.filter(lambda x: x > 10).map(lambda x: x * 10)
```

■ Comparison Table

| Feature | `map()` | `filter()` |
|----------------|-------------------------|---------------------------------------|
| Function | Transforms each element | Selects elements based on a condition |
| Output size | Same as input | ≤ input size |
| Transformation | Narrow | Narrow |

| Feature | `map()` | `filter()` |
|----------|-----------------------|----------------------|
| Use case | Modify or enrich data | Clean or subset data |

? What About `flatMap()`?

If you're wondering, `flatMap()` is like `map()` but flattens the output (used when one input element results in multiple output elements).

Would you like an example in **DataFrame API** as well?



Great question! In **PySpark**, **pivot** and **unpivot** are key operations when transforming data from **long to wide** (pivot) or **wide to long** (unpivot) formats — essential in data wrangling, reporting, and analytics.

1. Pivot in PySpark

Transforms rows into columns.

Use Case:

Turn values in one column into **distinct column headers**, and aggregate another column's values under them.

🔽 Example: Pivot

📤 Output:

```
+----+
| name|English| Math|
+----+
| Hari| 90| 85|
|Kishore| 88| 78|
+----+
```

2. Unpivot in PySpark (No direct function — use `stack()`)

■ Transforms columns back into rows.

Example: Unpivot

```
from pyspark.sql.functions import expr

# Starting with pivoted DataFrame (pivot_df)
unpivot_df = pivot_df.selectExpr(
        "name",
        "stack(2, 'Math', Math, 'English', English) as (subject, marks)"
)
unpivot_df.show()
```

📤 Output:

```
+----+
| name| subject|marks|
+----+
| Hari| Math| 85|
| Hari| English| 90|
|Kishore| Math| 78|
|Kishore| English| 88|
+----+
```


Operation Function (DataFrame API) Purpose Pivot groupBy().pivot().agg() Rows \rightarrow Columns (wide format) Unpivot selectExpr("stack(...)") Columns \rightarrow Rows (long format)

| Would you like an example where pivoting is used in a real reporting or BI scenario (e.g., sales per month per region)? | | |
|---|--|--|
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |