Spark SQL – DataFrames

Why Spark SQL?

It provides:

- Structured APIs (DataFrame)
- SQL query support
- Optimized execution engine
- Integration with multiple storage systems

Spark SQL is the structured data processing engine inside Apache Spark.

It enables:

- Relational-style processing
- Schema-aware transformations
- Automatic query optimization
- BI tool compatibility

Spark SQL Overview

Spark SQL allows you to:

- Query structured data using SQL
- Use DataFrame APIs
- Process semi-structured data (JSON, nested fields)
- Integrate with Hive Metastore
- Leverage Catalyst optimizer

In PySpark:

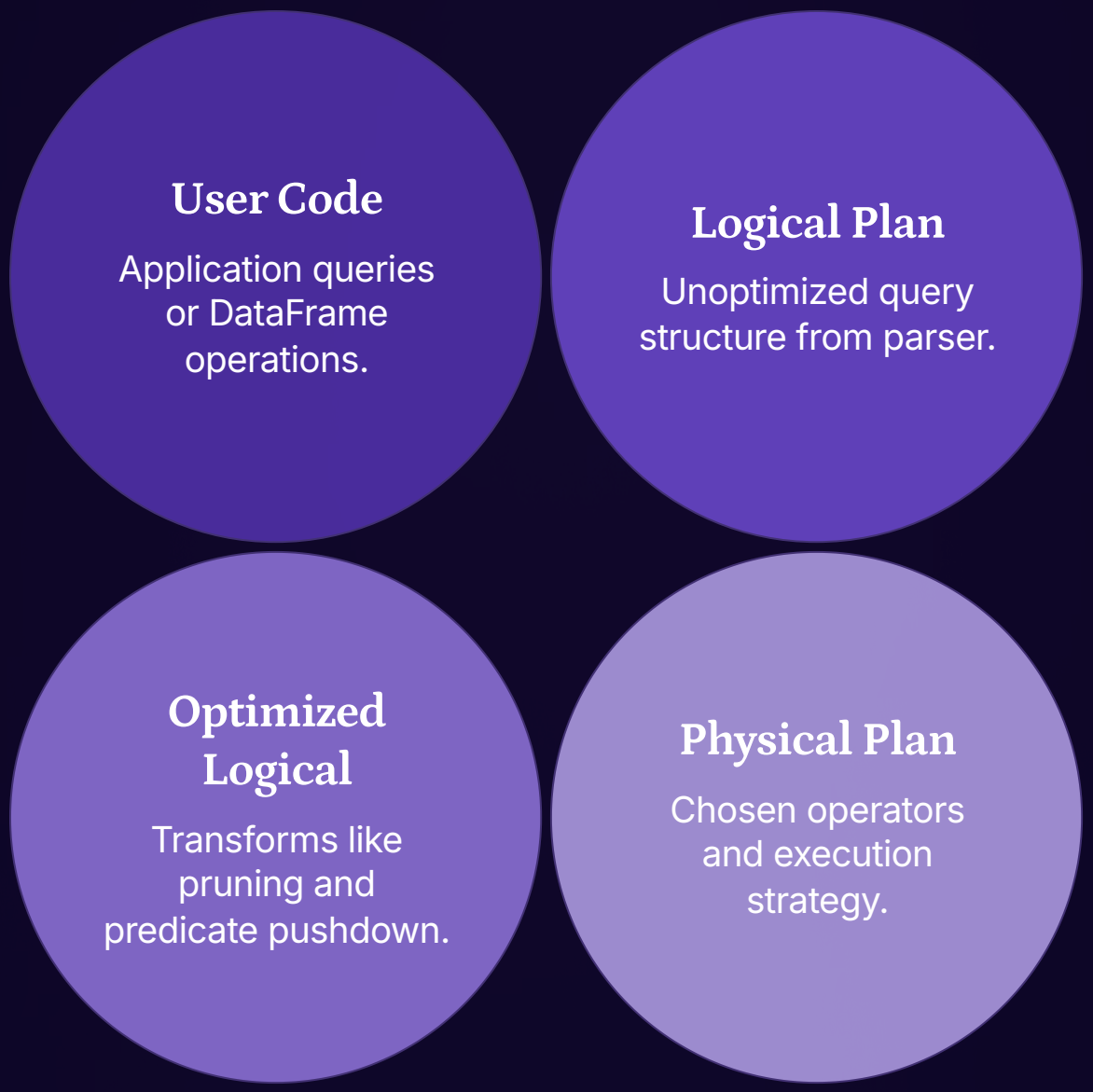
DataFrame is the primary structured API

Dataset API is not separately available

Spark SQL Architecture

1	2	3
Data Source API JSON, CSV, Parquet, ORC, Avro, JDBC	**Catalyst Optimizer** • Logical plan optimization • Predicate pushdown • Column pruning • Constant folding	**Tungsten Execution Engine** • Efficient memory management • Whole-stage code generation

Execution Flow:



The execution pipeline transforms user code through a series of optimization stages before final execution.

DataFrame Overview

A DataFrame is:

- A distributed collection of data
- Organized into named columns
- Schema-aware
- Immutable
- Similar to a relational table

Key Characteristics:

- Lazy evaluation
- Declarative transformations
- Optimized execution

Example:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder \
 .appName("example") \
 .getOrCreate()
df = spark.read.json("data.json")
df.printSchema()
df.show()
```

Dataset Overview (PySpark Context)

In Spark (Scala/Java):

- Dataset = Type-safe distributed collection
- Combines RDD and DataFrame benefits
- Uses Encoders

In PySpark:

- No separate Dataset API
- DataFrame = Dataset[Row]
- No compile-time type safety

Important: In Python, you work with DataFrames only.

DataFrame vs Dataset

Feature Comparison:

Feature	DataFrame (PySpark)	Dataset (Scala/Java)
Type Safety	No	Yes
Compile-time checks	No	Yes
Language Support	Python, Scala, Java	Scala, Java
Encoders Required	No	Yes

In PySpark → Use DataFrame API.

When to Use DataFrames (PySpark)

Use DataFrames when:

- Working in Python
- Processing structured or semi-structured data
- Needing SQL-like transformations
- Performance optimization is required
- Working with BI tools

Use RDD only when:

- Low-level transformations are required
- Custom partition-level logic is necessary

Type-Safe Transformations (Conceptual)

Scala Example:

```
case class Person(name: String, age: Int)
val ds = spark.read
  .json("data.json")
  .as[Person]
```

In PySpark:

- No compile-time schema enforcement
- Enforce schema manually

Example:

```
from pyspark.sql.types import (
 StructType, StructField,
 StringType, IntegerType
)
schema = StructType([
 StructField("name", StringType(), True),
 StructField("age", IntegerType(), True)
])
df = spark.read.schema(schema).json("data.json")
```

Encoders (Concept Awareness)

Encoders:

- Convert JVM objects into Spark's internal binary format
- Required for Datasets in Scala

Not applicable in PySpark.

Important for awareness in mixed-language teams.

Loading Data – Read APIs

Common Formats:

- JSON
- CSV
- Parquet
- ORC
- Avro

Common Options:

- header
- inferSchema
- delimiter
- mode (PERMISSIVE, DROPMALFORMED, FAILFAST)

Examples:

```
df_json = spark.read.json("data.json")

df_csv = spark.read \
 .option("header", True) \
 .option("inferSchema", True) \
 .csv("data.csv")

df_parquet = spark.read.parquet("data.parquet")
```

Writing Data

Write Examples:

```
df.write.mode("overwrite").parquet("output/")
df.write.mode("append").json("output_json/")
```

Partitioned Writes:

```
df.write.partitionBy("country").parquet("output/")
```

Write Modes:

overwrite	append
ignore	error

Column Expressions

Column operations are declarative.

Select & Alias:

```
from pyspark.sql.functions import col, when

df.select(
    col("name"),
    (col("salary") * 1.1).alias("updated_salary")
)
```

Conditional Column:

```
df.withColumn(
    "category",
    when(col("salary") > 50000, "High")
    .otherwise("Low")
)
```

Transformations build logical plans **(lazy execution)**.

Aggregations

GroupBy Aggregation:

```
from pyspark.sql.functions import avg, sum, count

df.groupBy("department") \
    .agg(
        avg("salary").alias("avg_salary"),
        count("*").alias("total")
    )
```

Advanced Aggregations:

- rollup()
- cube()
- Window functions

User Defined Functions (UDF)

Use UDF only when built-in functions are insufficient.

```
from pyspark.sql.functions import udf, col
from pyspark.sql.types import StringType

def categorize(age):
    return "Adult" if age >= 18 else "Minor"

categorize_udf = udf(categorize, StringType())

df.withColumn(
    "age_group",
    categorize_udf(col("age"))
)
```

Important:

- UDFs reduce optimization capability
- Prefer built-in Spark functions

Working with Nested Data

Example JSON:

```
{
  "name": "John",
  "address": {
    "city": "Chennai",
    "zip": "600001"
  }
}
```

Access Nested Fields:

```
df.select("address.city")
```

Explode Arrays:

```
from pyspark.sql.functions import explode

df.select(explode("items"))
```

JDBC Integration

Read from Database:

```
df = spark.read.format("jdbc") \
    .option("url", "jdbc:postgresql://host:5432/db") \
    .option("dbtable", "employees") \
    .option("user", "user") \
    .option("password", "pass") \
    .load()
```

Write to Database:

```
df.write.format("jdbc") \
    .option("url", "jdbc:postgresql://host:5432/db") \
    .option("dbtable", "output_table") \
    .mode("append") \
    .save()
```

Best Practice: Use `partitionColumn` for parallel reads.

Loading from Distributed Storage

HDFS

```
spark.read.parquet(
  "hdfs://namenode:8020/data/"
)
```

Amazon S3

```
spark.read.parquet(
  "s3a://bucket/path/"
)
```

Google Cloud Storage

```
spark.read.parquet(
  "gs://bucket/path/"
)
```

Azure Blob Storage

```
spark.read.parquet(
  "wasbs://container@account
  .blob.core.windows.net/path/"
)
```

Requires: Proper credentials, IAM roles or service accounts.

Performance Considerations

Best Practices:

1. Prefer Parquet or ORC over JSON/CSV
2. Avoid excessive UDF usage
3. Use column pruning
4. Partition data wisely
5. Cache only when reused
6. Minimize wide shuffles

Check Execution Plan:

```
df.explain(True)
```

Key Takeaways

Core Engine

Spark SQL is the structured processing engine of Apache Spark

Primary API

In PySpark, DataFrame is the primary API

Dataset API

Dataset API is Scala/Java specific

Schema

Schema definition improves reliability

Functions

Built-in functions outperform UDFs

Storage

Parquet is the preferred storage format

Optimizer

Catalyst optimizer automatically optimizes queries