PySpark

The Complete Textbook

From Zero to Production-Ready

By

Pavan Kola

A Simple, Hands-On Guide for Data Engineers

50+ Pages of Code Examples, Diagrams & Best Practices

2026 Edition

Table of Contents

Chapter 1: What is PySpark?

1.1 The Big Picture

Imagine you have a massive dataset - say 500 GB of user clickstream data. Your laptop has 16 GB RAM. How do you process it? That is exactly the problem PySpark solves.

PySpark is the Python API for Apache Spark, a distributed computing engine. Instead of processing data on one machine, Spark splits the work across a cluster of machines, each handling a portion of the data in parallel.

> **Think of it like this:**
> If you need to sort 1 million books, you could do it alone (single machine) or hire 100 people, give each person 10,000 books, have them sort their pile, then merge the results. That is distributed computing.

1.2 Why PySpark Over Pandas?

Pandas is fantastic for small to medium datasets that fit in memory. But when your data grows beyond what a single machine can handle, you need PySpark. Here is a direct comparison:

Feature	Pandas	PySpark
Data Size	Up to ~10 GB (single machine RAM)	Petabytes (distributed across cluster)
Processing	Single machine, single core	Distributed across multiple machines
Execution	Eager (runs immediately)	Lazy (builds plan, runs when needed)
Use Case	Data analysis, small ETL	Big data ETL, ML at scale
Learning Curve	Easier for beginners	Requires understanding of distributed concepts
Speed (small data)	Faster (no overhead)	Slower (cluster overhead)
Speed (big data)	Crashes or very slow	Scales linearly with cluster size

1.3 Spark Architecture - Simplified

Understanding Spark's architecture helps you write better code. Here are the key components:

Driver Program: This is your main program. It is the brain that coordinates everything. When you write spark.read.csv(...), the Driver creates a plan for how to execute it.

Cluster Manager: Think of this as the HR department. It manages resources (CPU, memory) across the cluster. Common options: YARN (Hadoop), Kubernetes, Mesos, or Standalone.

Executors: These are the workers. Each executor runs on a separate machine and processes a chunk of your data. They do the actual computation.

Tasks: The smallest unit of work. Each executor runs multiple tasks in parallel. If you have 100 partitions of data and 10 executors, each executor handles ~10 tasks.

```
# Simplified flow:
# 1. You write:  df.filter(col('age') > 30).groupBy('city').count()
# 2. Driver creates execution plan (DAG)
# 3. Cluster Manager allocates executors
# 4. Data is split into partitions across executors
# 5. Each executor processes its partitions
# 6. Results are collected back to the Driver
```

1.4 Lazy Evaluation - The Key Concept

This is the single most important concept in PySpark. Unlike Pandas, PySpark does NOT execute your code line by line. Instead, it builds up a plan (called a DAG - Directed Acyclic Graph) and only executes when you explicitly ask for a result.

Transformations (Lazy): Operations that define what to do but do NOT execute. Examples: filter(), select(), groupBy(), join(), withColumn(). These just add steps to the plan.

Actions (Trigger Execution): Operations that force Spark to execute the plan and return results. Examples: show(), count(), collect(), write(). These kick off actual computation.

```
# None of these lines execute anything yet:
df = spark.read.csv('huge_file.csv', header=True)
filtered = df.filter(col('age') > 25)
grouped = filtered.groupBy('department').count()

# THIS triggers execution of the entire chain above:
grouped.show()  # Now Spark reads, filters, groups, and displays
```

> **Best Practice:** Lazy evaluation lets Spark optimize your entire pipeline before running it. It might reorder operations, skip unnecessary columns, or push filters down to read less data. This is called the Catalyst Optimizer.

Chapter 2: Setting Up PySpark

2.1 Installation Methods

There are several ways to get PySpark running. Here are the most common approaches, from simplest to production-ready:

Method 1: pip install (Simplest)

```
# Install PySpark with pip
pip install pyspark

# Verify installation
python -c "import pyspark; print(pyspark.__version__)"
```

Method 2: Conda (Data Science teams)

```
conda install -c conda-forge pyspark
```

Method 3: AWS EMR / Databricks / Glue (Production)

In production, you rarely install Spark yourself. Cloud platforms like AWS EMR, Databricks, and AWS Glue come with PySpark pre-configured. You just write your code and submit it.

2.2 Creating a SparkSession

Every PySpark program starts with creating a SparkSession. This is your entry point to all Spark functionality.

```
from pyspark.sql import SparkSession

# Basic SparkSession
spark = SparkSession.builder \
    .appName('MyFirstApp') \
    .getOrCreate()

# SparkSession with configurations
spark = SparkSession.builder \
    .appName('ProductionApp') \
    .config('spark.sql.shuffle.partitions', '200') \
    .config('spark.executor.memory', '4g') \
    .config('spark.driver.memory', '2g') \
    .config('spark.sql.adaptive.enabled', 'true') \
```

```
    .getOrCreate()
```

Pro Tip:
In production, most configurations are set at cluster level (spark-submit) rather than in code. Hardcoding configs makes your code less portable. Use code-level config only for application-specific settings like shuffle partitions.

2.3 SparkSession vs SparkContext

In older Spark versions (before 2.0), you used SparkContext. SparkSession is the modern, unified entry point that combines SparkContext, SQLContext, and HiveContext into one.

```
# Old way (Spark 1.x) - DON'T use this anymore
from pyspark import SparkContext
sc = SparkContext('local', 'OldApp')

# New way (Spark 2.0+) - USE this
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('NewApp').getOrCreate()

# You can still access SparkContext if needed:
sc = spark.sparkContext
```

2.4 Running PySpark Locally vs Cluster

When developing, you typically run PySpark in local mode on your laptop. When deploying, you run it on a cluster. The code stays the same - only the configuration changes.

Mode	Config	When to Use
Local (1 core)	.master('local')	Quick testing
Local (all cores)	.master('local[*]')	Local development
YARN	.master('yarn')	Hadoop clusters
Kubernetes	.master('k8s://...')	K8s deployments
Standalone	.master('spark://...')	Dedicated Spark cluster

Best Practice: Always develop and test with a small sample of your data locally before running on the full dataset in the cluster. This saves time and cluster resources.

Chapter 3: RDDs - The Foundation

3.1 What is an RDD?

RDD stands for Resilient Distributed Dataset. It is the fundamental data structure in Spark. Think of it as an immutable, distributed collection of objects that can be processed in parallel.

Resilient: If a partition of data is lost (machine fails), Spark can recompute it from the original source using the lineage graph.

Distributed: Data is split across multiple machines in the cluster.

Dataset: A collection of records (rows of data).

3.2 Creating RDDs

```
from pyspark import SparkContext
sc = spark.sparkContext

# Method 1: From a Python collection
data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
rdd = sc.parallelize(data, numSlices=4)  # 4 partitions

# Method 2: From a file
text_rdd = sc.textFile('hdfs:///data/logs.txt')

# Method 3: From another RDD (transformation)
squared_rdd = rdd.map(lambda x: x ** 2)
```

3.3 Common RDD Operations

RDD operations fall into two categories: Transformations (lazy) and Actions (trigger execution).

Transformations (Lazy)

Operation	Description	Example
map()	Apply function to each element	rdd.map(lambda x: x * 2)
filter()	Keep elements matching condition	rdd.filter(lambda x: x > 5)
flatMap()	Map then flatten results	rdd.flatMap(lambda x: x.split(' '))
distinct()	Remove duplicates	rdd.distinct()

union()	Combine two RDDs	rdd1.union(rdd2)
reduceByKey()	Aggregate by key	rdd.reduceByKey(lambda a,b: a+b)
sortByKey()	Sort by key	rdd.sortByKey()
groupByKey()	Group values by key	rdd.groupByKey()

Actions (Trigger Execution)

Operation	Description	Example
collect()	Return all elements as list	rdd.collect()
count()	Count elements	rdd.count()
first()	Return first element	rdd.first()
take(n)	Return first n elements	rdd.take(5)
reduce()	Aggregate all elements	rdd.reduce(lambda a,b: a+b)
saveAsTextFile()	Write to file	rdd.saveAsTextFile('output/')
foreach()	Apply function (no return)	rdd.foreach(print)

3.4 Word Count - The Classic Example

```
# The 'Hello World' of distributed computing
text_rdd = sc.textFile('book.txt')

word_counts = (
    text_rdd
    .flatMap(lambda line: line.split(' '))    # Split lines into words
    .map(lambda word: (word.lower(), 1))      # Create (word, 1) pairs
    .reduceByKey(lambda a, b: a + b)          # Sum counts per word
    .sortBy(lambda x: x[1], ascending=False)  # Sort by count
)

# Display top 10 words
for word, count in word_counts.take(10):
    print(f'{word}: {count}')
```

Warning: In modern PySpark, you should prefer DataFrames over RDDs for almost everything. DataFrames are faster (Catalyst optimizer), easier to use, and provide better interoperability with SQL. RDDs are still useful for unstructured data or when you need fine-grained control.

Chapter 4: DataFrames - Your Daily Driver

4.1 What is a DataFrame?

A DataFrame is a distributed collection of data organized into named columns, similar to a table in a relational database or a Pandas DataFrame. It is the primary data structure you will use in PySpark.

Feature	RDD	DataFrame
Structure	Unstructured / any Python object	Tabular with named columns & types
Optimization	No automatic optimization	Catalyst optimizer + Tungsten engine
Ease of Use	Requires lambdas & manual logic	SQL-like API, very intuitive
Performance	Slower (Python serialization)	Much faster (optimized execution plan)
Use Case	Unstructured data, custom logic	Structured/semi-structured data (95% of use cases)

4.2 Creating DataFrames

From Python Collections

```
from pyspark.sql import SparkSession
from pyspark.sql.types import *

spark = SparkSession.builder.appName('DataFrames').getOrCreate()

# Method 1: From list of tuples
data = [('Alice', 28, 'Engineering'), ('Bob', 35, 'Marketing'),
        ('Charlie', 42, 'Engineering'), ('Diana', 31, 'Sales')]

df = spark.createDataFrame(data, ['name', 'age', 'department'])
df.show()

# Output:
# +-------+---+-----------+
# |   name|age| department|
# +-------+---+-----------+
# |  Alice| 28|Engineering|
# |    Bob| 35|  Marketing|
# |Charlie| 42|Engineering|
# |  Diana| 31|      Sales|
```

```
# +-------+---+-----------+
```

With Explicit Schema

```
# Defining schema explicitly (recommended for production)
schema = StructType([
    StructField('name', StringType(), True),
    StructField('age', IntegerType(), True),
    StructField('department', StringType(), True),
    StructField('salary', DoubleType(), True),
])

data = [('Alice', 28, 'Engineering', 95000.0),
        ('Bob', 35, 'Marketing', 82000.0)]

df = spark.createDataFrame(data, schema)
df.printSchema()

# root
#  |-- name: string (nullable = true)
#  |-- age: integer (nullable = true)
#  |-- department: string (nullable = true)
#  |-- salary: double (nullable = true)
```

Best Practice: Always define schemas explicitly in production code. Schema inference reads extra data and can guess types incorrectly (e.g., reading '001' as integer 1 instead of string '001').

4.3 Reading External Data

```
# CSV
df = spark.read.csv('data.csv', header=True, inferSchema=True)
df = spark.read.option('header', True).schema(my_schema).csv('data.csv')

# JSON
df = spark.read.json('data.json')
df = spark.read.option('multiLine', True).json('nested.json')

# Parquet (preferred format for big data)
df = spark.read.parquet('data.parquet')

# Delta Lake
df = spark.read.format('delta').load('delta_table/')

# JDBC (Database)
df = spark.read.format('jdbc') \
    .option('url', 'jdbc:postgresql://host:5432/db') \
    .option('dbtable', 'public.users') \
    .option('user', 'admin') \
    .option('password', 'secret') \
    .load()
```

4.4 Writing Data

```
# Write modes: 'overwrite', 'append', 'ignore', 'error'

# Parquet (columnar, compressed - best for analytics)
df.write.mode('overwrite').parquet('output/data.parquet')

# Partitioned write (crucial for performance)
df.write.mode('overwrite') \
    .partitionBy('year', 'month') \
    .parquet('output/partitioned/')

# CSV
df.write.mode('overwrite') \
    .option('header', True) \
    .csv('output/data.csv')

# Single file output (use with caution on large data)
df.coalesce(1).write.mode('overwrite').csv('output/single/')
```

Warning: Never use coalesce(1) on large datasets. It forces all data onto one partition (one machine), defeating the purpose of distributed computing and likely causing an OutOfMemoryError.

Chapter 5: DataFrame Operations

5.1 Selecting Columns

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

# Multiple ways to select columns
df.select('name', 'age')                    # By name (string)
df.select(col('name'), col('age'))           # Using col()
df.select(df.name, df.age)                   # Using df.column
df.select(df['name'], df['age'])             # Using df['column']

# Select with transformation
df.select(
    col('name'),
    col('salary'),
    (col('salary') * 0.1).alias('bonus')     # Calculated column
)
```

5.2 Filtering Rows

```
from pyspark.sql.functions import col

# Single condition
df.filter(col('age') > 30)
df.filter(df.age > 30)           # Alternative syntax
df.where(col('age') > 30)        # where() is alias for filter()

# Multiple conditions (AND)
df.filter((col('age') > 25) & (col('department') == 'Engineering'))

# Multiple conditions (OR)
df.filter((col('age') > 40) | (col('salary') > 100000))

# NOT condition
df.filter(~col('name').startswith('A'))

# IN condition
df.filter(col('department').isin('Engineering', 'Sales'))

# NULL checks
df.filter(col('email').isNotNull())
df.filter(col('phone').isNull())

# String patterns
df.filter(col('name').like('%son'))
df.filter(col('name').contains('Ali'))
df.filter(col('email').rlike('^[a-z]+@.*\.com$'))  # Regex
```

Important:
Always wrap multiple filter conditions in parentheses. Due to Python operator precedence, (col('a') > 5) & (col('b') < 10) is correct, but col('a') > 5 & col('b') < 10 will produce an error.

5.3 Adding & Modifying Columns

```
from pyspark.sql.functions import col, lit, when, upper, concat

# Add new column
df = df.withColumn('bonus', col('salary') * 0.1)

# Add constant column
df = df.withColumn('country', lit('USA'))

# Modify existing column
df = df.withColumn('name', upper(col('name')))

# Conditional column (CASE WHEN)
df = df.withColumn('level',
    when(col('salary') > 100000, 'Senior')
    .when(col('salary') > 70000, 'Mid')
    .otherwise('Junior')
)

# Concatenate columns
df = df.withColumn('full_info',
    concat(col('name'), lit(' - '), col('department'))
)

# Rename column
df = df.withColumnRenamed('name', 'employee_name')

# Drop columns
df = df.drop('temp_col', 'debug_col')
```

5.4 Sorting

```
from pyspark.sql.functions import col, asc, desc

# Simple sort
df.orderBy('salary')                          # Ascending (default)
df.orderBy(col('salary').desc())               # Descending
df.sort('salary')                              # sort() = orderBy()

# Multi-column sort
df.orderBy(
    col('department').asc(),
    col('salary').desc()                       # Within each dept
```

```
)
```

Warning: Sorting is expensive in distributed systems because data from all partitions must be compared. Avoid unnecessary sorts, especially before a write operation where order does not matter.

Chapter 6: Aggregations & GroupBy

6.1 Simple Aggregations

```
from pyspark.sql.functions import count, sum, avg, min, max,
    countDistinct, approx_count_distinct, stddev, variance

# Aggregate entire DataFrame
df.select(
    count('*').alias('total_rows'),
    countDistinct('department').alias('unique_depts'),
    avg('salary').alias('avg_salary'),
    max('salary').alias('max_salary'),
    min('salary').alias('min_salary'),
    sum('salary').alias('total_payroll'),
    stddev('salary').alias('salary_stddev'),
).show()
```

6.2 GroupBy Aggregations

```
# Group by single column
df.groupBy('department').agg(
    count('*').alias('employee_count'),
    avg('salary').alias('avg_salary'),
    max('salary').alias('max_salary'),
).show()

# Group by multiple columns
df.groupBy('department', 'level').agg(
    count('*').alias('count'),
    sum('salary').alias('total_salary'),
).orderBy('department', 'level').show()

# Quick aggregations without agg()
df.groupBy('department').count().show()
df.groupBy('department').sum('salary').show()
df.groupBy('department').avg('salary').show()
```

6.3 Pivot Tables

Pivot tables rotate rows into columns, which is very useful for creating summary reports.

```
# Pivot: departments as rows, quarters as columns
pivot_df = df.groupBy('department') \
    .pivot('quarter', ['Q1', 'Q2', 'Q3', 'Q4']) \
    .agg(sum('revenue'))
```

```
# Output:
# +-----------+------+------+------+------+
# | department|    Q1|    Q2|    Q3|    Q4|
# +-----------+------+------+------+------+
# |Engineering|150000|180000|200000|175000|
# |  Marketing| 80000| 95000|110000| 90000|
# |      Sales|200000|250000|220000|280000|
# +-----------+------+------+------+------+
```

Best Practice: Always specify the values list in pivot() like .pivot('col', ['val1', 'val2']). Without it, Spark makes an extra pass over the data to discover unique values, which is expensive on large datasets.

6.4 Rollup and Cube

Rollup and Cube are advanced grouping operations that create subtotals and grand totals automatically.

```
# Rollup: Hierarchical subtotals (left to right)
df.rollup('department', 'level') \
    .agg(count('*').alias('count'), sum('salary').alias('total')) \
    .orderBy('department', 'level') \
    .show()
# Shows: grand total, department totals, department+level combos

# Cube: All possible combinations of subtotals
df.cube('department', 'level') \
    .agg(count('*').alias('count')) \
    .show()
# Shows: grand total, dept totals, level totals, dept+level combos
```

Chapter 7: Joins

7.1 Join Types Explained

Joins combine two DataFrames based on a common column. Understanding join types is critical for getting correct results.

Join Type	What it Returns	Use Case
inner	Only matching rows from both sides	When you need only records that exist in both tables
left (left_outer)	All rows from left + matching from right	Keep all source records, enrich with lookup data
right (right_outer)	All rows from right + matching from left	Rarely used; restructure as left join instead
full (full_outer)	All rows from both sides	Reconciliation: find what exists in either source
left_semi	Left rows that have a match in right (no right columns)	Filter left table by existence in right table
left_anti	Left rows with NO match in right	Find missing records, gap analysis
cross	Every combination (cartesian product)	Generate all possible pairs (use carefully)

7.2 Join Syntax

```
# Sample DataFrames
employees = spark.createDataFrame([
    (1, 'Alice', 101), (2, 'Bob', 102), (3, 'Charlie', 101),
    (4, 'Diana', 103), (5, 'Eve', None)
], ['emp_id', 'name', 'dept_id'])

departments = spark.createDataFrame([
    (101, 'Engineering'), (102, 'Marketing'),
    (104, 'Finance')
], ['dept_id', 'dept_name'])

# Inner Join
employees.join(departments, 'dept_id', 'inner').show()

# Left Join
employees.join(departments, 'dept_id', 'left').show()

# Join on different column names
```

```
employees.join(
    departments,
    employees.dept_id == departments.dept_id,
    'inner'
).drop(departments.dept_id)  # Remove duplicate column

# Multi-column join
df1.join(df2, ['col_a', 'col_b'], 'inner')

# Complex join condition
df1.join(df2,
    (df1.id == df2.id) & (df1.date >= df2.start_date),
    'inner'
)
```

7.3 Join Performance & Optimization

Joins are one of the most expensive operations in distributed computing. Here are the key strategies:

Broadcast Join (Small + Large table)

When one table is small (under ~10 MB), Spark can broadcast it to all executors, avoiding an expensive shuffle.

```
from pyspark.sql.functions import broadcast

# Explicit broadcast hint
result = large_df.join(
    broadcast(small_lookup_df),
    'join_key'
)

# Spark auto-broadcasts tables under threshold
# Default: spark.sql.autoBroadcastJoinThreshold = 10MB
spark.conf.set('spark.sql.autoBroadcastJoinThreshold', '50m')
```

> **Best Practice:** Always broadcast dimension/lookup tables when joining with fact tables. A 50 MB broadcast avoids shuffling 500 GB of fact data across the network.

Sort-Merge Join (Large + Large table)

For large-to-large joins, Spark uses Sort-Merge Join. Both tables are sorted by the join key and then merged. This requires a shuffle (data redistribution), which is expensive.

Handling Skewed Joins

Data skew happens when some join keys have far more records than others (e.g., 90% of orders belong to 10 customers). This creates a bottleneck where one executor does most of the work.

```
# Enable Adaptive Query Execution (handles skew automatically)
spark.conf.set('spark.sql.adaptive.enabled', 'true')
spark.conf.set('spark.sql.adaptive.skewJoin.enabled', 'true')

# Manual approach: salt the key
from pyspark.sql.functions import lit, rand, floor, concat

salt_range = 10
salted_big = big_df.withColumn('salt', floor(rand() * salt_range))
salted_big = salted_big.withColumn('salted_key',
    concat(col('join_key'), lit('_'), col('salt')))
```

Chapter 8: Window Functions

8.1 What are Window Functions?

Window functions perform calculations across a set of rows that are related to the current row. Unlike GROUP BY, window functions do NOT collapse rows - you keep all your original rows and add the computed value as a new column.

> **Think of it like this:**
> GROUP BY is like summarizing a report (1 row per group). Window functions are like adding a summary column to every row of your detailed report.

8.2 Defining a Window

```
from pyspark.sql.window import Window
from pyspark.sql.functions import row_number, rank, dense_rank,
    lag, lead, sum, avg, count, col

# Window partitioned by department, ordered by salary
dept_window = Window.partitionBy('department') \
    .orderBy(col('salary').desc())

# Window with frame specification
running_window = Window.partitionBy('department') \
    .orderBy('date') \
    .rowsBetween(Window.unboundedPreceding, Window.currentRow)

# Window without partition (entire DataFrame)
global_window = Window.orderBy(col('salary').desc())
```

8.3 Ranking Functions

```
# ROW_NUMBER: Unique sequential number (1, 2, 3, 4)
df = df.withColumn('row_num',
    row_number().over(dept_window))

# RANK: Same rank for ties, gaps after (1, 2, 2, 4)
df = df.withColumn('rank',
    rank().over(dept_window))

# DENSE_RANK: Same rank for ties, no gaps (1, 2, 2, 3)
df = df.withColumn('dense_rank',
    dense_rank().over(dept_window))

# Common pattern: Get top N per group
```

```
top3_per_dept = (
    df.withColumn('rn', row_number().over(dept_window))
      .filter(col('rn') <= 3)
      .drop('rn')
)
```

8.4 Analytical Functions

```
window = Window.partitionBy('department').orderBy('date')

# LAG: Access previous row's value
df = df.withColumn('prev_salary', lag('salary', 1).over(window))

# LEAD: Access next row's value
df = df.withColumn('next_salary', lead('salary', 1).over(window))

# Month-over-month change
df = df.withColumn('salary_change',
    col('salary') - lag('salary', 1).over(window))

# Percentage change
df = df.withColumn('pct_change',
    ((col('salary') - lag('salary', 1).over(window))
     / lag('salary', 1).over(window) * 100))
```

8.5 Running Totals & Moving Averages

```
# Running total within each department
running = Window.partitionBy('department') \
    .orderBy('date') \
    .rowsBetween(Window.unboundedPreceding, Window.currentRow)

df = df.withColumn('running_total', sum('revenue').over(running))

# 7-day moving average
moving_7d = Window.partitionBy('department') \
    .orderBy('date') \
    .rowsBetween(-6, Window.currentRow)  # Current + 6 prior

df = df.withColumn('moving_avg_7d', avg('revenue').over(moving_7d))

# Cumulative percentage
cum_window = Window.partitionBy('department') \
    .orderBy(col('salary').desc()) \
    .rowsBetween(Window.unboundedPreceding, Window.currentRow)

df = df.withColumn('cum_pct',
    sum('salary').over(cum_window) / sum('salary').over(
        Window.partitionBy('department'))  # Total for department
)
```

Best Practice: Window functions are much more efficient than self-joins for computing row-relative values (previous value, running total, rank). Always prefer window functions over self-joins.

Chapter 9: PySpark SQL

9.1 Using SQL with PySpark

PySpark lets you write SQL queries directly against DataFrames. This is useful if you are more comfortable with SQL or when translating existing SQL queries to PySpark.

```
# Register DataFrame as a temporary view
df.createOrReplaceTempView('employees')

# Now you can query it with SQL
result = spark.sql('''
    SELECT department,
           COUNT(*) as employee_count,
           AVG(salary) as avg_salary,
           MAX(salary) as max_salary
    FROM employees
    WHERE age > 25
    GROUP BY department
    HAVING COUNT(*) > 5
    ORDER BY avg_salary DESC
''')

result.show()
```

9.2 Temp Views vs Global Temp Views

```
# Temp view: visible only in current SparkSession
df.createOrReplaceTempView('my_table')

# Global temp view: visible across all SparkSessions
df.createOrReplaceGlobalTempView('shared_table')

# Access global temp view (note the prefix)
spark.sql('SELECT * FROM global_temp.shared_table')
```

9.3 Complex SQL Queries

```
# Window functions in SQL
spark.sql('''
    SELECT name, department, salary,
        RANK() OVER (PARTITION BY department ORDER BY salary DESC)
            AS dept_rank,
        salary - LAG(salary) OVER (PARTITION BY department
            ORDER BY hire_date) AS salary_diff
    FROM employees
```

```
''')

# CTEs (Common Table Expressions)
spark.sql('''
    WITH dept_stats AS (
        SELECT department,
               AVG(salary) as avg_sal,
               STDDEV(salary) as std_sal
        FROM employees
        GROUP BY department
    )
    SELECT e.name, e.salary, d.avg_sal,
           (e.salary - d.avg_sal) / d.std_sal AS z_score
    FROM employees e
    JOIN dept_stats d ON e.department = d.department
    WHERE ABS((e.salary - d.avg_sal) / d.std_sal) > 2
''')
```

SQL or DataFrame API?

Both produce the same execution plan under the hood. Use whichever your team prefers. Many teams use SQL for ad-hoc analysis and the DataFrame API for production ETL pipelines where composability and testability matter more.

Chapter 10: Handling Missing Data

10.1 Understanding NULLs in PySpark

Missing data is represented as NULL in PySpark (similar to None in Python). Handling NULLs correctly is critical because they can silently affect your results.

> **Warning:** NULL propagates in computations. If salary is NULL, then salary * 2 is also NULL, and AVG(salary) skips NULL rows entirely. This can give misleading results if not handled properly.

10.2 Detecting NULLs

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

# Count NULLs per column
df.select([
    count(when(col(c).isNull(), c)).alias(c)
    for c in df.columns
]).show()

# Count NULLs and NaNs (for numeric columns)
df.select([
    count(when(col(c).isNull() | isnan(c), c)).alias(c)
    for c in df.columns
]).show()

# Filter rows with NULLs in specific column
df.filter(col('email').isNull()).show()
df.filter(col('email').isNotNull()).show()
```

10.3 Handling NULLs

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

# Drop rows with any NULL
df.dropna()                              # Any column has NULL
df.dropna(how='all')                     # All columns are NULL
df.dropna(subset=['email', 'phone'])     # NULL in specific cols
df.dropna(thresh=3)                      # Keep if >= 3 non-NULLs

# Fill NULLs with values
df.fillna(0)                             # Fill all with 0
df.fillna({'salary': 0, 'dept': 'Unknown', 'age': 30})

# Coalesce: first non-NULL value
```

```
df.withColumn('contact',
    coalesce(col('email'), col('phone'), lit('No Contact')))

# Replace with conditional logic
df.withColumn('salary',
    when(col('salary').isNull(),
         avg('salary').over(Window.partitionBy('department')))
    .otherwise(col('salary')))
```

Best Practice: Fill NULLs based on business context. For numeric columns, consider using the group average (as shown above) rather than 0. For categorical columns, use a meaningful default like 'Unknown' rather than an empty string.

Chapter 11: Schema & Data Types

11.1 PySpark Data Types

Type	Python Equivalent	Example
StringType()	str	'hello'
IntegerType()	int	42
LongType()	int (large)	9999999999
FloatType()	float (32-bit)	3.14
DoubleType()	float (64-bit)	3.14159265359
BooleanType()	bool	True / False
DateType()	datetime.date	2024-01-15
TimestampType()	datetime.datetime	2024-01-15 14:30:00
ArrayType()	list	[1, 2, 3]
MapType()	dict	{'key': 'value'}
StructType()	namedtuple	Row(name='Alice', age=30)
DecimalType(p,s)	Decimal	DecimalType(10, 2) for money

11.2 Defining Complex Schemas

```
from pyspark.sql.types import *

# Nested schema (e.g., JSON data)
schema = StructType([
    StructField('user_id', LongType(), False),     # NOT nullable
    StructField('name', StringType(), True),
    StructField('address', StructType([            # Nested struct
        StructField('street', StringType(), True),
        StructField('city', StringType(), True),
        StructField('zip', StringType(), True),
    ]), True),
    StructField('tags', ArrayType(StringType()), True),  # Array
    StructField('metadata', MapType(                     # Map
        StringType(), StringType()), True),
])
```

11.3 Casting & Type Conversion

```
from pyspark.sql.functions import col, to_date, to_timestamp
```

```
# Cast column type
df = df.withColumn('age', col('age').cast('integer'))
df = df.withColumn('salary', col('salary').cast(DoubleType()))

# String to Date
df = df.withColumn('date', to_date(col('date_str'), 'yyyy-MM-dd'))

# String to Timestamp
df = df.withColumn('ts',
    to_timestamp(col('ts_str'), 'yyyy-MM-dd HH:mm:ss'))

# Check schema after transformations
df.printSchema()
df.dtypes  # Returns list of (column_name, type_string) tuples
```

Warning: When casting fails (e.g., casting 'abc' to integer), PySpark returns NULL silently instead of throwing an error. Always validate your data after type conversions.

Chapter 12: User Defined Functions (UDFs)

12.1 When to Use UDFs

UDFs let you apply custom Python logic to DataFrame columns. However, they come with a significant performance cost because data must be serialized from the JVM to Python and back.

Approach	Performance	When to Use
Built-in functions	Best (runs in JVM)	Always try this first
Pandas UDF (vectorized)	Good (uses Arrow)	Custom logic on batches
Regular UDF	Worst (row-by-row serialization)	Last resort

12.2 Regular UDFs

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

# Method 1: Decorator
@udf(returnType=StringType())
def categorize_age(age):
    if age is None:
        return 'Unknown'
    elif age < 25:
        return 'Young'
    elif age < 45:
        return 'Mid-Career'
    else:
        return 'Senior'

df = df.withColumn('age_group', categorize_age(col('age')))

# Method 2: Register inline
clean_phone = udf(lambda x: x.replace('-','').replace(' ','')
                  if x else None, StringType())

df = df.withColumn('clean_phone', clean_phone(col('phone')))
```

12.3 Pandas UDFs (Vectorized) - Much Faster

Pandas UDFs process data in batches using Apache Arrow, making them 10-100x faster than regular UDFs.

```
import pandas as pd
from pyspark.sql.functions import pandas_udf
from pyspark.sql.types import DoubleType, StringType

# Series to Series (most common)
@pandas_udf(DoubleType())
def normalize(series: pd.Series) -> pd.Series:
    return (series - series.mean()) / series.std()

df = df.withColumn('salary_normalized', normalize(col('salary')))

# Grouped Map (apply function to each group)
@pandas_udf(df.schema, functionType=PandasUDFType.GROUPED_MAP)
def fill_missing_by_group(pdf: pd.DataFrame) -> pd.DataFrame:
    pdf['salary'] = pdf['salary'].fillna(pdf['salary'].median())
    return pdf

result = df.groupBy('department').apply(fill_missing_by_group)
```

Best Practice: Rule of thumb: 1) Try built-in functions first. 2) If custom logic is needed, use Pandas UDFs. 3) Only use regular UDFs when the logic is too complex for either option.

Chapter 13: Working with Dates & Strings

13.1 Date Functions

```
from pyspark.sql.functions import (
    current_date, current_timestamp, date_add, date_sub,
    datediff, months_between, year, month, dayofmonth,
    dayofweek, hour, minute, second, date_format,
    to_date, to_timestamp, unix_timestamp, from_unixtime,
    last_day, next_day, trunc, date_trunc
)

# Current date/time
df.withColumn('today', current_date())
df.withColumn('now', current_timestamp())

# Date arithmetic
df.withColumn('next_week', date_add(col('date'), 7))
df.withColumn('last_month', date_sub(col('date'), 30))
df.withColumn('days_diff', datediff(col('end'), col('start')))
df.withColumn('months_diff',
    months_between(col('end'), col('start')))

# Extract components
df.withColumn('yr', year(col('date')))
df.withColumn('mon', month(col('date')))
df.withColumn('dow', dayofweek(col('date')))  # 1=Sun, 7=Sat

# Format dates
df.withColumn('formatted',
    date_format(col('date'), 'MMM dd, yyyy'))  # Jan 15, 2024

# Truncate to period
df.withColumn('month_start', trunc(col('date'), 'month'))
df.withColumn('year_start', trunc(col('date'), 'year'))
```

13.2 String Functions

```
from pyspark.sql.functions import (
    col, upper, lower, trim, ltrim, rtrim, lpad, rpad,
    length, substring, split, concat, concat_ws,
    regexp_extract, regexp_replace, translate,
    initcap, reverse, instr
)

# Case functions
df.withColumn('upper_name', upper(col('name')))
df.withColumn('title_name', initcap(col('name')))
```

```
# Trim whitespace
df.withColumn('clean', trim(col('input')))

# Substring
df.withColumn('first3', substring(col('name'), 1, 3))

# Split string into array
df.withColumn('parts', split(col('full_name'), ' '))
df.withColumn('first', split(col('full_name'), ' ')[0])

# Concatenate
df.withColumn('full', concat(col('first'), lit(' '), col('last')))
df.withColumn('csv', concat_ws(',', col('a'), col('b'), col('c')))

# Regex
df.withColumn('domain',
    regexp_extract(col('email'), '@(.+)', 1))
df.withColumn('clean_phone',
    regexp_replace(col('phone'), '[^0-9]', ''))

# Padding (useful for fixed-width formats)
df.withColumn('id_padded', lpad(col('id'), 10, '0'))
```

Chapter 14: Working with Complex Types

14.1 Arrays

```
from pyspark.sql.functions import (
    array, array_contains, explode, explode_outer,
    posexplode, size, array_distinct, array_union,
    array_intersect, array_except, sort_array, flatten
)

# Create array column
df = df.withColumn('skills', array(lit('Python'), lit('SQL')))

# Check if array contains value
df.filter(array_contains(col('tags'), 'urgent'))

# Explode: one row per array element
df.select('name', explode('tags').alias('tag'))  # Drops NULLs
df.select('name', explode_outer('tags').alias('tag'))  # Keeps NULLs

# Array operations
df.withColumn('num_tags', size(col('tags')))
df.withColumn('unique_tags', array_distinct(col('tags')))
df.withColumn('sorted', sort_array(col('tags')))
```

14.2 Structs (Nested Objects)

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

# Access nested fields (dot notation)
df.select(col('address.city'), col('address.zip'))

# Create struct column
df = df.withColumn('location',
    struct(col('city'), col('state'), col('zip')))

# Flatten struct (bring nested fields to top level)
df = df.select('*',
    col('address.street').alias('street'),
    col('address.city').alias('city'),
).drop('address')
```

14.3 Maps

```
from pyspark.sql.functions import col, create_map, map_keys,
    map_values, element_at, explode
```

```
# Create map from columns
df = df.withColumn('info',
    create_map(lit('name'), col('name'), lit('age'), col('age')))

# Access map values
df.withColumn('name_val', element_at(col('info'), 'name'))
df.withColumn('keys', map_keys(col('info')))
df.withColumn('values', map_values(col('info')))

# Explode map into key-value rows
df.select('id', explode('metadata')).show()
# Produces columns: id, key, value
```

Flattening JSON:

When reading deeply nested JSON, use explode() to flatten arrays and dot notation to access struct fields. Process layer by layer from the outside in.

Chapter 15: Performance Optimization

15.1 Understanding Shuffles

A shuffle is when Spark redistributes data across the cluster. It involves disk I/O, network I/O, and serialization - the three most expensive operations. Shuffles are triggered by operations like groupBy, join, distinct, orderBy, and repartition.

```
# Operations that cause shuffles:
df.groupBy('key').count()         # Shuffle: data grouped by key
df1.join(df2, 'key')              # Shuffle: both DFs by key
df.distinct()                     # Shuffle: detect duplicates
df.orderBy('col')                # Shuffle: global sort
df.repartition(100)              # Shuffle: redistribute

# Operations that do NOT shuffle:
df.filter(col('age') > 30)       # Map operation (per-partition)
df.select('name', 'age')         # Map operation
df.withColumn('x', col('a')*2)   # Map operation
df.coalesce(50)                  # Reduce partitions (no shuffle)
```

15.2 Partitioning Strategy

Number of Partitions

```
# Check current partitions
print(df.rdd.getNumPartitions())

# Rules of thumb:
# - 2-4 partitions per CPU core in cluster
# - Each partition: 128 MB - 256 MB of data
# - Too few partitions = underutilized cluster
# - Too many partitions = excessive overhead

# Repartition (causes shuffle - use when increasing)
df = df.repartition(200)
df = df.repartition(200, 'key_col')  # Hash partition by key

# Coalesce (no shuffle - use when decreasing)
df = df.coalesce(50)

# Set default shuffle partitions
spark.conf.set('spark.sql.shuffle.partitions', '200')
```

Data Partitioning on Disk

```
# Partition by frequently filtered columns
df.write.mode('overwrite') \
    .partitionBy('year', 'month') \
    .parquet('output/sales/')

# Result: output/sales/year=2024/month=01/part-00000.parquet
#         output/sales/year=2024/month=02/part-00000.parquet

# When reading, Spark skips irrelevant partitions
# This query only reads year=2024, month=01 files:
spark.read.parquet('output/sales/') \
    .filter((col('year') == 2024) & (col('month') == 1))
```

Best Practice: Choose partition columns with low cardinality (year, month, country). High cardinality columns (user_id) create too many tiny files, which hurts read performance. Aim for partition files of 128 MB to 1 GB.

15.3 Caching & Persistence

```
from pyspark import StorageLevel

# Cache in memory (most common)
df.cache()          # = persist(StorageLevel.MEMORY_AND_DISK)

# Persist with specific storage level
df.persist(StorageLevel.MEMORY_ONLY)       # Memory only
df.persist(StorageLevel.MEMORY_AND_DISK)    # Spill to disk
df.persist(StorageLevel.DISK_ONLY)          # Disk only
df.persist(StorageLevel.MEMORY_ONLY_SER)    # Serialized (compact)

# IMPORTANT: cache() is lazy - data is cached on first action
df.cache()     # Just marks for caching
df.count()     # NOW it actually caches

# Release cache
df.unpersist()

# Check what is cached
spark.catalog.isCached('my_table')
```

When to Cache	When NOT to Cache
DataFrame used multiple times	DataFrame used only once
After expensive computation	Small DataFrames (read is fast)
Before iterative algorithms	When memory is limited
Lookup tables used in joins	Before a single write operation

Chapter 16: Adaptive Query Execution (AQE)

16.1 What is AQE?

Adaptive Query Execution (AQE) is one of the most impactful features in modern Spark (3.0+). It optimizes queries at runtime based on actual data statistics, not just estimates. Enable it and Spark gets significantly smarter.

```
# Enable AQE (enabled by default in Spark 3.2+)
spark.conf.set('spark.sql.adaptive.enabled', 'true')

# Key AQE features:
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')
```

16.2 AQE Features

Coalescing Shuffle Partitions

Without AQE, you set spark.sql.shuffle.partitions to a fixed number (default 200). Too few causes OOM; too many creates tiny partitions with high overhead. AQE automatically merges small post-shuffle partitions into larger ones.

Optimizing Skew Joins

When a join key has heavily skewed data (some keys have millions of rows, others have few), AQE detects this at runtime and splits the large partitions into smaller sub-partitions that can be processed in parallel.

Converting Sort-Merge to Broadcast Join

If AQE discovers at runtime that one side of a join is small enough to broadcast, it automatically switches from Sort-Merge Join to Broadcast Join - even if the initial estimate was wrong.

> **Best Practice:** If you are running Spark 3.0 or later, always enable AQE. It handles many performance problems automatically that previously required manual tuning.

Chapter 17: File Formats Deep Dive

17.1 Format Comparison

Format	Type	Compression	Best For
CSV	Row-based, Text	Poor	Data exchange, small files, human-readable
JSON	Row-based, Text	Poor	API data, semi-structured, nested data
Parquet	Columnar, Binary	Excellent	Analytics, data warehousing (most common)
ORC	Columnar, Binary	Excellent	Hive ecosystem, write-heavy workloads
Avro	Row-based, Binary	Good	Streaming, schema evolution, write-heavy
Delta Lake	Columnar (Parquet+)	Excellent	ACID transactions, time-travel, upserts
Iceberg	Columnar (Parquet+)	Excellent	Multi-engine, schema evolution, partitioning

17.2 Why Parquet?

Parquet is the gold standard for analytical workloads. Here is why:

Columnar Storage: Data is stored by column, not by row. When your query selects only 5 out of 100 columns, Spark reads only those 5 columns from disk - not the entire dataset.

Efficient Compression: Similar values in a column compress much better than mixed values in a row. Parquet files are typically 50-90% smaller than CSV.

Predicate Pushdown: Parquet files store min/max statistics per column chunk. If you filter WHERE age > 30 and a chunk has max(age) = 25, Spark skips that entire chunk without reading it.

Schema Embedded: The schema is stored in the file itself, so you never need to specify it when reading.

```
# Writing Parquet with compression
df.write.mode('overwrite') \
    .option('compression', 'snappy') \
    .parquet('output/data.parquet')
```

```
# Compression options: snappy (default, balanced),
#                      gzip (smaller, slower),
#                      zstd (best ratio),
#                      lz4 (fastest)
```

17.3 Delta Lake & Apache Iceberg

Modern table formats add database-like features on top of Parquet files. They solve the limitations of raw Parquet:

Feature	Raw Parquet	Delta Lake / Iceberg
ACID Transactions	No	Yes - safe concurrent writes
UPDATE/DELETE/MERGE	No - must rewrite entire partition	Yes - efficient row-level operations
Time Travel	No	Yes - query any historical version
Schema Evolution	Manual and risky	Built-in with safety checks
Small File Compaction	Manual	Automatic optimization

```
# Delta Lake example
# Write
df.write.format('delta').mode('overwrite').save('delta_table/')

# Read specific version (time travel)
spark.read.format('delta').option('versionAsOf', 5).load('delta_table/')

# MERGE (upsert) - incredibly useful for CDC pipelines
from delta.tables import DeltaTable

delta_table = DeltaTable.forPath(spark, 'delta_table/')
delta_table.alias('target').merge(
    updates_df.alias('source'),
    'target.id = source.id'
).whenMatchedUpdateAll() \
 .whenNotMatchedInsertAll() \
 .execute()
```

Chapter 18: Error Handling & Debugging

18.1 Common Errors & Fixes

Error	Cause	Fix
OutOfMemoryError	Data too large for executor memory	Increase executor memory or reduce partition size
AnalysisException	Column not found, wrong SQL	Check column names with df.columns, fix SQL syntax
Py4JJavaError	JVM-side error (type mismatch, etc.)	Read the Java exception message for root cause
Task serialization error	UDF references non-serializable object	Use broadcast variables or restructure code
Skew in stage	Uneven data distribution	Salt keys or enable AQE skew join handling
Too many small files	Over-partitioned writes	Coalesce before writing or use repartition

18.2 Debugging Strategies

```
# 1. Check the execution plan
df.explain()            # Simple plan
df.explain(True)        # Full physical + logical plan
df.explain('formatted') # Pretty-printed plan

# 2. Look for shuffles in the plan
# Exchange = shuffle (expensive!)
# BroadcastExchange = broadcast (good!)

# 3. Sample your data for debugging
small_df = df.sample(fraction=0.01)  # 1% sample
small_df = df.limit(1000)            # First 1000 rows

# 4. Check data at each step
df_step1 = df.filter(col('age') > 25)
print(f'After filter: {df_step1.count()} rows')

df_step2 = df_step1.groupBy('dept').count()
df_step2.show()

# 5. Monitor in Spark UI (localhost:4040)
# - Stages tab: look for skewed tasks
# - SQL tab: see query plans
# - Storage tab: check cached data
```

18.3 Production Error Handling

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

logger = logging.getLogger('etl_pipeline')

def safe_read(spark, path, schema):
    """Read with error handling"""
    try:
        df = spark.read.schema(schema).parquet(path)
        row_count = df.count()
        logger.info(f'Read {row_count} rows from {path}')
        if row_count == 0:
            logger.warning(f'Empty dataset at {path}')
        return df
    except Exception as e:
        logger.error(f'Failed to read {path}: {str(e)}')
        raise

def validate_data(df, name):
    """Basic data quality checks"""
    total = df.count()
    nulls = df.filter(col('id').isNull()).count()
    dupes = total - df.dropDuplicates(['id']).count()

    logger.info(f'{name}: {total} rows, {nulls} nulls, {dupes} dupes')

    if nulls / total > 0.1:  # >10% null IDs
        raise ValueError(f'{name}: Too many NULL IDs ({nulls}/{total})')
    return df
```

Chapter 19: Real-World ETL Pipeline

19.1 End-to-End Pipeline Example

Let us build a complete ETL pipeline that processes e-commerce order data. This example demonstrates patterns you will use daily as a Data Engineer.

Step 1: Define Schemas

```
from pyspark.sql import SparkSession
from pyspark.sql.types import *
from pyspark.sql.functions import *

spark = SparkSession.builder \
    .appName('EcommerceETL') \
    .config('spark.sql.adaptive.enabled', 'true') \
    .getOrCreate()

# Source schemas
orders_schema = StructType([
    StructField('order_id', StringType(), False),
    StructField('customer_id', StringType(), False),
    StructField('order_date', StringType(), True),
    StructField('total_amount', DoubleType(), True),
    StructField('status', StringType(), True),
])

items_schema = StructType([
    StructField('order_id', StringType(), False),
    StructField('product_id', StringType(), False),
    StructField('quantity', IntegerType(), True),
    StructField('unit_price', DoubleType(), True),
])
```

Step 2: Extract (Read Source Data)

```
def extract(spark, orders_path, items_path):
    """Read raw data from source"""
    orders_raw = spark.read.schema(orders_schema) \
        .option('header', True) \
        .csv(orders_path)

    items_raw = spark.read.schema(items_schema) \
        .option('header', True) \
        .csv(items_path)

    return orders_raw, items_raw
```

Step 3: Transform (Clean & Enrich)

```
def transform(orders_raw, items_raw):
    """Clean, validate, and enrich data"""

    # Clean orders
    orders_clean = (
        orders_raw
        .withColumn('order_date',
            to_date(col('order_date'), 'yyyy-MM-dd'))
        .withColumn('total_amount',
            when(col('total_amount') < 0, lit(0.0))
            .otherwise(col('total_amount')))
        .filter(col('order_id').isNotNull())
        .dropDuplicates(['order_id'])
    )

    # Calculate item-level metrics
    items_agg = (
        items_raw
        .withColumn('line_total',
            col('quantity') * col('unit_price'))
        .groupBy('order_id')
        .agg(
            sum('line_total').alias('calc_total'),
            count('*').alias('item_count'),
            sum('quantity').alias('total_quantity'),
        )
    )

    # Join and enrich
    enriched = (
        orders_clean
        .join(items_agg, 'order_id', 'left')
        .withColumn('year', year(col('order_date')))
        .withColumn('month', month(col('order_date')))
        .withColumn('order_size',
            when(col('total_quantity') > 10, 'Large')
            .when(col('total_quantity') > 3, 'Medium')
            .otherwise('Small'))
    )

    return enriched
```

Step 4: Load (Write Output)

```
def load(df, output_path):
    """Write processed data partitioned by year/month"""
    (
        df
        .repartition('year', 'month')
        .write
        .mode('overwrite')
```

```
        .partitionBy('year', 'month')
        .parquet(output_path)
    )
    print(f'Written {df.count()} records to {output_path}')

# Run the pipeline
orders_raw, items_raw = extract(
    spark, 's3://bucket/raw/orders/', 's3://bucket/raw/items/')
enriched = transform(orders_raw, items_raw)
load(enriched, 's3://bucket/processed/orders/')
```

Best Practice: In production, wrap each step in try/except blocks, add logging, implement data quality checks between steps, and use orchestrators like Apache Airflow to manage dependencies and retries.

Chapter 20: Slowly Changing Dimensions

20.1 What are SCDs?

In data warehousing, Slowly Changing Dimensions (SCDs) describe how to handle changes to dimension data over time. For example, when a customer changes their address, how do you store both the old and new address?

Type	Strategy	Example
SCD Type 1	Overwrite the old value	Customer address updated in-place. History is lost.
SCD Type 2	Add a new row with versioning	New row with new address, old row marked as inactive. Full history preserved.
SCD Type 3	Add columns for current + previous	Columns: current_address, previous_address. Limited history.

20.2 SCD Type 2 Implementation

SCD Type 2 is the most common pattern in data engineering. Here is a complete PySpark implementation:

```
from pyspark.sql.functions import *
from pyspark.sql.window import Window

def apply_scd2(existing_df, updates_df, key_col, tracked_cols):
    """Apply SCD Type 2 logic"""

    today = current_date()
    max_date = to_date(lit('9999-12-31'))

    # Find changed records
    changes = existing_df.alias('e').join(
        updates_df.alias('u'), key_col, 'inner'
    ).filter(
        # At least one tracked column changed
        reduce(
            lambda a, b: a | b,
            [col(f'e.{c}') != col(f'u.{c}') for c in tracked_cols]
        ) & (col('e.is_current') == True)
    ).select(col(f'e.{key_col}'))

    # Close old records (set end_date and is_current=False)
```

```
    closed = existing_df.join(
        changes, key_col, 'left_semi'
    ).filter(col('is_current') == True
    ).withColumn('end_date', today
    ).withColumn('is_current', lit(False))

    # Create new current records
    new_records = updates_df.join(
        changes, key_col, 'left_semi'
    ).withColumn('start_date', today
    ).withColumn('end_date', max_date
    ).withColumn('is_current', lit(True))

    # Unchanged records
    unchanged = existing_df.join(
        changes, key_col, 'left_anti')

    # Net new records (not in existing)
    net_new = updates_df.join(
        existing_df.select(key_col).distinct(), key_col, 'left_anti'
    ).withColumn('start_date', today
    ).withColumn('end_date', max_date
    ).withColumn('is_current', lit(True))

    return unchanged.unionByName(closed) \
        .unionByName(new_records).unionByName(net_new)
```

Chapter 21: PySpark with AWS

21.1 PySpark on AWS Services

Service	What it is	Best For
EMR	Managed Spark cluster	Full control, custom configs, complex pipelines
Glue	Serverless Spark	Simple ETL, no cluster management, pay per use
Databricks on AWS	Managed Spark platform	Best developer experience, Delta Lake native
Athena	SQL on S3 (not Spark)	Ad-hoc queries, no infrastructure

21.2 Reading/Writing to S3

```
# SparkSession with S3 access
spark = SparkSession.builder \
    .appName('S3Pipeline') \
    .config('spark.hadoop.fs.s3a.access.key', 'AKIAIOSFODNN7EXAMPLE') \
    .config('spark.hadoop.fs.s3a.secret.key', 'wJalrXUtnFEMI/K7MDENG') \
    .config('spark.hadoop.fs.s3a.impl',
            'org.apache.hadoop.fs.s3a.S3AFileSystem') \
    .getOrCreate()

# Read from S3
df = spark.read.parquet('s3a://my-bucket/data/input/')

# Write to S3
df.write.mode('overwrite') \
    .partitionBy('year', 'month') \
    .parquet('s3a://my-bucket/data/output/')
```

IAM Roles are Better:

In production, never hardcode AWS credentials. Use IAM roles attached to your EMR cluster or Glue job. The Spark session will automatically use the role credentials.

21.3 AWS Glue ETL Job Example

```
import sys
from awsglue.transforms import *
from awsglue.utils import getResolvedOptions
from awsglue.context import GlueContext
from pyspark.context import SparkContext
```

```
# Initialize
args = getResolvedOptions(sys.argv, ['JOB_NAME'])
sc = SparkContext()
glueContext = GlueContext(sc)
spark = glueContext.spark_session

# Read from Glue Catalog
source = glueContext.create_dynamic_frame.from_catalog(
    database='my_database',
    table_name='raw_orders'
)

# Convert to DataFrame for PySpark operations
df = source.toDF()

# Apply transformations
processed = df.filter(col('status') != 'cancelled') \
    .withColumn('order_date', to_date(col('order_date')))

# Write back
glueContext.write_dynamic_frame.from_options(
    frame=DynamicFrame.fromDF(processed, glueContext, 'output'),
    connection_type='s3',
    connection_options={'path': 's3://bucket/processed/'},
    format='parquet'
)
```

Chapter 22: Testing PySpark Code

22.1 Why Test PySpark Code?

PySpark pipelines process millions of records in production. A bug in your transformation logic can corrupt downstream reports and dashboards. Testing catches these bugs before they reach production.

22.2 Unit Testing Transformations

```
import pytest
from pyspark.sql import SparkSession
from pyspark.sql.functions import col

# Shared SparkSession for all tests
@pytest.fixture(scope='session')
def spark():
    return SparkSession.builder \
        .master('local[2]') \
        .appName('Tests') \
        .getOrCreate()

# Test your transformation function
def test_categorize_orders(spark):
    # Arrange: create test data
    data = [(1, 15.0), (2, 75.0), (3, 200.0)]
    df = spark.createDataFrame(data, ['id', 'amount'])

    # Act: apply transformation
    result = categorize_orders(df)  # Your function

    # Assert: check results
    rows = result.collect()
    assert rows[0]['category'] == 'Small'   # 15.0
    assert rows[1]['category'] == 'Medium'  # 75.0
    assert rows[2]['category'] == 'Large'   # 200.0

def test_null_handling(spark):
    data = [(1, None), (2, 50.0)]
    df = spark.createDataFrame(data, ['id', 'amount'])

    result = categorize_orders(df)

    null_row = result.filter(col('id') == 1).collect()[0]
    assert null_row['category'] == 'Unknown'
```

22.3 Testing Patterns

Pattern	What to Test	Example
Happy Path	Normal data produces correct output	Valid orders are enriched correctly
Edge Cases	NULLs, empty strings, zero values	NULL amount categorized as 'Unknown'
Data Quality	Counts, nulls, duplicates after transform	No duplicate order_ids after dedup
Schema	Output has expected columns and types	output.schema matches expected_schema
Business Rules	Complex logic produces correct results	Discounts applied correctly by tier

Best Practice: Test with small, handcrafted datasets where you know the exact expected output. This makes tests fast, deterministic, and easy to debug when they fail.

Chapter 23: PySpark Best Practices

23.1 Code Organization

```
# Recommended project structure for PySpark ETL:

# my_project/
#   |-- config/
#   |     |-- settings.py          # Spark configs, paths
#   |-- src/
#   |     |-- extract.py           # Data readers
#   |     |-- transform.py         # Business logic
#   |     |-- load.py              # Data writers
#   |     |-- utils.py             # Shared helpers
#   |-- tests/
#   |     |-- test_transform.py    # Unit tests
#   |     |-- test_integration.py  # Integration tests
#   |-- jobs/
#   |     |-- daily_etl.py         # Main entry point
#   |-- requirements.txt
#   |-- README.md
```

23.2 Performance Do's and Don'ts

Do	Don't
Use built-in functions (col, when, etc.)	Use Python UDFs for simple operations
Filter early to reduce data volume	Filter after expensive operations like joins
Broadcast small lookup tables	Let Spark sort-merge join a 1MB table with a 1TB table
Use Parquet format	Use CSV for large analytical datasets
Define schemas explicitly	Rely on inferSchema for production
Cache DataFrames used multiple times	Cache every intermediate DataFrame
Use partitionBy for time-series writes	Write everything to a single file
Enable AQE	Manually tune everything without AQE first
Use coalesce() to reduce partitions	Use repartition(1) to create single files
Use select() to pick needed columns	Carry 100 columns through entire pipeline

23.3 Common Pitfalls

Pitfall 1: Calling .collect() on large DataFrames

collect() pulls ALL data to the driver. On a 100 GB DataFrame, this will crash your driver with an OutOfMemoryError.

```
# BAD - never collect large data
all_data = huge_df.collect()  # Driver OOM!

# GOOD - use show(), take(), or write
huge_df.show(20)           # Only fetches 20 rows
sample = huge_df.take(100) # Only fetches 100 rows
huge_df.write.parquet('output/')  # Stays distributed
```

Pitfall 2: Using Python loops instead of Spark operations

```
# BAD - defeats the purpose of distributed computing
for row in df.collect():
    process(row)  # Sequential, single-machine processing

# GOOD - let Spark distribute the work
df.withColumn('result', my_udf(col('input')))
```

Pitfall 3: Ignoring data skew

If 80% of your orders are from 1% of your customers, a groupBy or join on customer_id will create massive partitions for those top customers while other executors sit idle.

```
# Check for skew
df.groupBy('customer_id').count() \
    .orderBy(col('count').desc()) \
    .show(10)

# If top customer has 10M rows and average is 100,
# you have a severe skew problem. Enable AQE or salt keys.
```

Chapter 24: Interview Quick Reference

This chapter provides a condensed reference of the most frequently asked PySpark concepts in data engineering interviews.

24.1 Key Concepts to Remember

Concept	One-Line Explanation
Lazy Evaluation	Transformations build a plan; actions trigger execution
DAG	Directed Acyclic Graph - Spark's execution plan
Catalyst Optimizer	Optimizes logical plan into efficient physical plan
Tungsten Engine	Manages memory and CPU at binary level for speed
Narrow Transformation	Each input partition maps to one output partition (map, filter)
Wide Transformation	Input partitions contribute to multiple outputs (groupBy, join) - causes shuffle
Shuffle	Data redistribution across cluster - most expensive operation
Partition	A chunk of data that fits on one executor
Broadcast Variable	Read-only data sent to all executors (small lookup tables)
Accumulator	Write-only shared variable for aggregating values across tasks
Checkpoint	Saves RDD to reliable storage, breaking lineage chain
Persist vs Cache	cache() = persist(MEMORY_AND_DISK). persist() lets you choose storage level
AQE	Adaptive Query Execution - runtime optimization (Spark 3.0+)

24.2 Spark Execution Model

```
# Job -> Stages -> Tasks

# Job: Triggered by an action (show, count, write)
# Stage: Separated by shuffles (wide transformations)
```

```
# Task: One partition processed by one executor core

# Example: df.filter().groupBy().count().show()
# Job 1 triggered by .show()
#   Stage 1: read + filter (narrow - no shuffle)
#   Stage 2: groupBy + count (wide - shuffle at boundary)

# Parallelism = min(partitions, executor_cores)
```

24.3 Commonly Asked Transformations

Task	PySpark Code
Remove duplicates	df.dropDuplicates(['col1', 'col2'])
Top N per group	row_number().over(Window.partitionBy('grp').orderBy(desc('val')))
Running total	sum('val').over(Window.orderBy('date').rowsBetween(unboundedPreceding, currentRow))
Fill NULLs by group	last('val', True).over(Window.partitionBy('grp').orderBy('date'))
Unpivot	stack(3, 'Q1', Q1, 'Q2', Q2, 'Q3', Q3)
Explode JSON array	explode(col('items'))
Flatten nested struct	select('*', 'address.*').drop('address')

24.4 Performance Tuning Checklist

Check	Command / Config
Enable AQE	spark.sql.adaptive.enabled = true
Right partition count	2-4x CPU cores, 128-256 MB each
Broadcast small tables	broadcast(small_df)
Use columnar format	Write as Parquet, not CSV
Filter early	Push filters before joins
Avoid UDFs	Use built-in functions first
Cache wisely	Only cache reused DataFrames
Check for skew	groupBy(key).count().orderBy(desc('count'))
Minimize shuffles	Fewer groupBys, pre-partition data
Select needed columns only	df.select('col1', 'col2') early in pipeline

Chapter 25: Cheat Sheet & Quick Reference

25.1 Import Essentials

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import (
    col, lit, when, otherwise,
    count, sum, avg, min, max, countDistinct,
    upper, lower, trim, concat, concat_ws, split,
    to_date, to_timestamp, year, month, dayofmonth, datediff,
    row_number, rank, dense_rank, lag, lead,
    explode, array_contains, size,
    coalesce, isnan, regexp_extract, regexp_replace,
    broadcast, struct, array, create_map,
    date_format, current_date, current_timestamp
)
from pyspark.sql.window import Window
from pyspark.sql.types import (
    StructType, StructField, StringType, IntegerType,
    LongType, DoubleType, BooleanType, DateType,
    TimestampType, ArrayType, MapType
)
```

25.2 SparkSession Quick Start

```
spark = SparkSession.builder \
    .appName('QuickStart') \
    .config('spark.sql.adaptive.enabled', 'true') \
    .config('spark.sql.shuffle.partitions', '200') \
    .getOrCreate()
```

25.3 Read/Write Cheat Sheet

Operation	Code
Read CSV	spark.read.option('header', True).schema(s).csv(path)
Read Parquet	spark.read.parquet(path)
Read JSON	spark.read.option('multiLine', True).json(path)
Write Parquet	df.write.mode('overwrite').parquet(path)
Partitioned Write	df.write.partitionBy('year','month').parquet(path)
Single File CSV	df.coalesce(1).write.option('header',True).csv(path)

25.4 DataFrame Operations Cheat Sheet

Task	Code
Select columns	df.select('a', 'b')
Filter rows	df.filter(col('x') > 5)
Add column	df.withColumn('new', col('a') * 2)
Rename column	df.withColumnRenamed('old', 'new')
Drop column	df.drop('col')
Sort	df.orderBy(col('x').desc())
Remove duplicates	df.dropDuplicates(['key'])
Group & aggregate	df.groupBy('g').agg(sum('v').alias('total'))
Join	df1.join(df2, 'key', 'left')
Union	df1.unionByName(df2, allowMissingColumns=True)
Window function	row_number().over(Window.partitionBy('g').orderBy('v'))
NULL handling	df.fillna({'col': 0}).dropna(subset=['key'])
Cast type	df.withColumn('x', col('x').cast('integer'))

- End of Textbook -

Happy Learning & Keep Building!