PySpark Notes
Section 1: Introduction

1.1 What is Apache Spark?

Apache Spark is an open-source, distributed computing framework designed for **large-scale data processing**. Unlike traditional single-machine systems, Spark distributes data and computation across a cluster, enabling horizontal scaling.

- **Key Distinction**:
  - Single-machine systems (e.g., Pandas) process data in-memory but hit hardware limits (RAM/CPU).
  - Spark processes data in parallel across nodes. For example, a 1GB dataset split into 4 partitions (250MB each) is processed simultaneously across 4 worker nodes.
- **Scalability**:
  - Scales from a single machine to thousands of nodes (e.g., AWS EMR, Databricks).
  - Handles petabytes of data by adding worker nodes.

1.2 Spark Architecture

Spark operates on a **master-slave model**:

- **Cluster Manager**:
  - Manages cluster resources (e.g., AWS EC2 instances).
  - Types: Standalone, YARN, Kubernetes, or Mesos.
- **Driver Node**:
  - Runs the SparkSession (entry point for PySpark).
  - Converts code into a Directed Acyclic Graph (DAG) of tasks.
- **Worker Nodes**:
  - Execute tasks in parallel.
  - Store partitions of Resilient Distributed Datasets (RDDs) or DataFrames.

1.3 Spark Benefits

- **In-Memory Computation**:
-

```
# Pandas (single-machine)
df = pd.read_csv("data.csv")  # Limited by RAM
# PySpark (distributed)
df = spark.read.csv("data.csv")  # Distributed across workers
```

  - Retains intermediate data in memory, avoiding disk I/O (10–100x faster than Hadoop MapReduce).
- **Lazy Evaluation**:
  - Transformations (e.g., filter, join) are queued until an action (e.g., count, show) is called.
  - Example:

```
# No execution happens here
filtered_df = df.filter(df["age"] > 30)
transformed_df = filtered_df.withColumn("age_plus_10", df["age"] + 10)
# Execution triggered by action
transformed_df.show()
```

- Spark optimizes the DAG before execution (e.g., predicate pushdown).
- **Fault Tolerance**:
  - Rebuilds lost data partitions using lineage (log of transformations).
- **Partitioning**:
  - Data split into partitions for parallel processing.
  - Example:

```
df.rdd.getNumPartitions()  # Returns partition count
df.repartition(8)  # Explicitly increase parallelism
```

1.4 Job Execution Hierarchy

- **Job**: Created per action (e.g., count, save).
- **Stage**: Created per shuffle (e.g., groupBy, join).
- **Task**: Operates on a single partition.

1.5 PySpark Language Options

- **Python (PySpark)**:
  - Pros: Easy syntax, integration with ML libraries (e.g., TensorFlow).
  - Cons: Slightly slower than Scala due to Python UDF overhead.
- **Scala**: Native Spark integration, optimal for performance-critical tasks.

1.6 Setting Up Databricks

Step-by-Step Guide:

1. Create a **Databricks Community Edition** account.
2. **Create a Cluster**:
   - Runtime: Choose "Latest LTS" (e.g., 13.3 LTS).
   - Worker Type: i3.xlarge (Community Edition has resource limits).
3. **Create a Notebook**:
   - Attach it to the cluster.
   - Run a test script:

```
data = [("Alice", 34), ("Bob", 45)]
df = spark.createDataFrame(data, ["name", "age"])
df.show()
```

Best Practices:

- Avoid collect(): It retrieves all data to the driver, risking OutOfMemory errors.
- Use .cache() judiciously: Only for DataFrames reused across multiple actions.

Common Pitfalls:

- **Skewed Partitions**:

```
# Bad: Causes uneven data distribution
df.groupBy("country").count()

# Better: Salt keys or use adaptive query execution (AQE) in Spark 3+
```

- **Over-Partitioning**: Excessive partitions increase task scheduling overhead.

Real-World Use Case:
A retail company processes 100TB of sales data daily using PySpark for:

- ETL pipelines (cleaning/aggregating data).
- Real-time analytics via Structured Streaming.

Section 2: Reading Data in PySpark (CSV & JSON)

2.1 Reading CSV Files

Core Mechanism

PySpark uses the spark.read API (part of the SparkSession entry point) to ingest structured/semi-structured data. For CSV files:

```
# Full parameter breakdown
df = spark.read.format("csv") \
  .option("inferSchema", "true") \  # Auto-detect data types (e.g., int vs string)
  .option("header", "true") \      # Use first row as headers
  .option("sep", ",") \           # Custom delimiter (e.g., ";" for European CSVs)
  .option("nullValue", "NA") \     # Treat "NA" as null
  .load("/path/to/file.csv")
```

Key Options Explained

- inferSchema:
  - Spark scans the entire dataset to infer column types (e.g., "123" → IntegerType).
  - **Cost**: Double read operation (slow for large datasets).
- header: If false, columns are named _c0, _c1, etc.
- mode:
  - PERMISSIVE (default): Corrupt records populate a _corrupt_record column.
  - DROPMALFORMED: Discard malformed rows.
  - FAILFAST: Abort on first corrupt record.
  -

Debugging Paths

```
# List files to verify path correctness
display(dbutils.fs.ls("/FileStore/tables/"))  # Databricks-specific
# Alternative in vanilla Spark
import os
print(os.listdir("/path/"))  # Local file system
```

2.2 Reading JSON Files

JSON Formats

- **Single-line**: One JSON object per line (common in logs).
- **Multi-line**: Pretty-printed JSON (nested objects across lines).

Code Example with Multi-line JSON

```
df = spark.read.format("json") \
  .option("multiLine", "true") \
  .option("primitivesAsString", "true") \  # Force numbers as strings
  .load("/path/to/nested_data.json")
```

Nested JSON Handling

- Spark automatically infers nested schemas as StructType or ArrayType.
- Example schema for nested JSON:

```
|-- name: string
 |-- address: struct
   |-- city: string
   |-- zip: integer
```

2.3 Schema Handling

Why Define Schemas Explicitly?

- Avoid schema inference overhead.
- Enforce data type consistency (e.g., prevent "price" from mixing strings and floats).

Method 1: DDL Schema (Simple)

```
ddl_schema = "item_id STRING, price DECIMAL(10,2), timestamp TIMESTAMP"
df = spark.read.schema(ddl_schema).csv("/path.csv")
```

- **Pros**: Concise syntax.
- **Cons**: Limited to flat schemas; no control over nullability.
-

Method 2: StructType (Advanced)

```
from pyspark.sql.types import *

schema = StructType([
  StructField("item_id", StringType(), nullable=False),  # Non-nullable
  StructField("price", DoubleType(), nullable=True),
  StructField("attributes", MapType(StringType(), StringType()))  # Nested type
])

df = spark.read.schema(schema).json("/path.json")
```

- **Pros**: Supports nested data (arrays/maps), enforces nullability.

Schema Override Example

```
# Force "item_weight" to string despite numeric values
schema = StructType([StructField("item_weight", StringType())])
df = spark.read.schema(schema).csv("bigmart_sales.csv")
```

2.4 Data Inspection

show() vs display()

- df.show(n=20, truncate=True, vertical=False):
  - PySpark-native.
  - truncate=False shows full column content.
- display(df):
  - Databricks-specific with UI enhancements (charts, filters).
  - Not available in vanilla Spark.

Schema Verification

```
df.printSchema()  # Output:
# root
#  |-- column1: integer (nullable = true)
#  |-- column2: string (nullable = true)
```

Best Practices

1. **Avoid** inferSchema **in Production**:
   - Predefine schemas for reliability and performance.
2. **Use Multi-line JSON Sparingly**:
   - Not splittable → single-task reads → memory bottlenecks.
3. **Handle Corrupt Records**:

```
spark.read.option("columnNameOfCorruptRecord", "_malformed") \
   .json("data.json")
```

Common Pitfalls

- **Schema Mismatches**:
  - Error: AnalysisException: Cannot up cast column from StringType to IntegerType.
  - Fix: Use .schema() or cast() during transformations.
- **Date Format Ambiguity**:

```
spark.read.option("dateFormat", "MM/dd/yyyy").csv("dates.csv")
```

Real-World Use Case

A financial institution ingests 10M-row CSV transaction logs daily:

- Explicit schema ensures "amount" is always DecimalType(38,2).
- mode="DROPMALFORMED" discards 0.1% invalid records.

Section 3: Basic PySpark Transformations

3.1 Select Transformation

Purpose

Extract specific columns or create derived columns while controlling schema evolution.

Syntax & Use Cases

```
# Method 1: String-based selection (simple but limited)
df.select("item_identifier", "item_weight")

# Method 2: Column objects (supports complex logic)
from pyspark.sql.functions import col, concat
df.select(
  col("item_identifier"),
  concat(col("item_type"), lit("_category")).alias("category")
)
```

Key Considerations

- Use col() for:
  - Column operations (e.g., col("A") + col("B")).
  - Ambiguous column references (e.g., after joins).
- Avoid select("*") in production to minimize data movement.

3.2 Alias Transformation

Purpose

Rename columns for clarity or to resolve naming conflicts (e.g., after joins).

Example with Joins

```
df1.alias("left").join(
  df2.alias("right"),
  col("left.id") == col("right.id")
)
```

Best Practices

- Use aliases in aggregations:

```
df.groupBy("category").agg(
  avg("price").alias("avg_price")
)
```

- Aliases are immutable: Original column remains unless explicitly overwritten.

3.3 Filter Transformation

Syntax Deep Dive

```
# Equivalent to SQL WHERE clause
df.filter(col("price") > 100)
df.where(col("price") > 100)  # Alias for filter()

# Complex logic
df.filter(
   (col("city") == "NYC") &
   (col("sales").isNotNull()) &
   (~col("product").isin("A", "B"))
)
```

Optimization Tips

- Push filters upstream: Apply filters before joins/aggregations to reduce data volume.
- Use isin() instead of multiple OR conditions for readability.

3.4 WithColumn Transformation

Purpose

Add new columns or modify existing ones without mutating the original DataFrame.

Advanced Use Cases

```
from pyspark.sql.functions import when

# Conditional logic
df = df.withColumn(
   "discount_tier",
   when(col("sales") > 1000, "Gold")
    .when(col("sales") > 500, "Silver")
    .otherwise("Bronze")
)

# Type-safe operations
from pyspark.sql.types import DateType
df = df.withColumn(
   "delivery_date",
   col("order_date").cast(DateType())
)
```

Performance Note

Chaining multiple withColumn calls creates a single optimized plan:

```
# Single pass over data (efficient)
df.withColumn(“A”, ...).withColumn(“B”, ...)
```

3.5 Type Casting

Common Scenarios

```
# Handle malformed data
from pyspark.sql.functions import try_cast
df.withColumn("weight", try_cast(col("weight_str"), DoubleType())

# Date conversions
df.withColumn(
  "order_date",
  to_date(col("date_string"), "MM/dd/yyyy")
)
```

Pitfalls

- Silent nulls: Invalid casts return null (use try_cast for error tracking).
- Locale issues: Date formats vary (e.g., dd/MM/yyyy vs MM/dd/yyyy).

3.6 Sorting (OrderBy)

Shuffle Implications

Sorting triggers a **wide transformation** (data shuffling across nodes).

Optimization Strategies

```
# Limit shuffle partitions
spark.conf.set("spark.sql.shuffle.partitions", 200)

# Use approximate sorting for large datasets
df.sort(col("timestamp").asc_nulls_first())
```

Top-N Patterns

```
from pyspark.sql.window import Window
window = Window.partitionBy("category").orderBy(col("sales").desc())

df.withColumn("rank", rank().over(window))
 .filter(col("rank") <= 3)  # Get top 3 per category
```

Best Practices

1. **Column Selection**:
    - Prefer explicit column lists over select("*").
2. **Filter Early**:
    - Apply filters before resource-intensive operations.
3. **Alias Conflicts**:
    - Use df.hint("broadcast") when joining large + small datasets.

Common Errors

- AnalysisException: Cannot resolve column name:
    - Caused by typos or unaliased duplicate columns.
- IllegalArgumentException: requirement failed:
    - Invalid type casting (e.g., string → integer with non-numeric values).

Real-World Use Case

An e-commerce platform uses these transformations to:

- Clean raw JSON clickstream data (filter, withColumn).
- Enrich user profiles (select, alias).
- Generate leaderboards (sort, window functions).

Section 4: Advanced PySpark Operations

4.1 GroupBy and Aggregations

Core Mechanism

groupBy triggers a **shuffle operation** to redistribute data by key. Aggregations then compute summaries per partition.

```
from pyspark.sql.functions import approx_count_distinct, collect_list

# Advanced aggregations
df.groupBy("category") \
  .agg(
    approx_count_distinct("user_id", 0.05).alias("unique_users"),  # 5% error tolerance
    collect_list("product_id").alias("products_purchased")
  )
```

Best Practices

- **Skew Mitigation**: Use salting for skewed keys:

```
df.withColumn("salted_key", concat(col("key"), lit("_"), (rand() * 10).cast("int")))
```

- **Pre-Aggregation**: Reduce data size with combineByKey before shuffling.

Pitfalls

- **OOM Errors**: Large groups (e.g., 100M users) → Use spill-to-disk config:

```
spark.conf.set("spark.sql.shuffle.partitions", 2000)
```

4.2 Joins

Join Types Explained

Join Type	Behavior	Use Case
inner	Intersection of keys	Fact-dimension linking
left_anti	Rows in left NOT in right	Data validation
left_semi	Rows in left that exist in right	Filtering

Optimization Techniques

- **Broadcast Join**:

```
# Auto-broadcast if table < spark.sql.autoBroadcastJoinThreshold (default 10MB)
df_large.join(broadcast(df_small), "key")
```

- **Sort-Merge Join**: For large tables with sorted keys:

```
df1.hint("sort").join(df2.hint("sort"), "key")
```

Skew Handling

```
# Split skewed keys into sub-ranges
df_skew = df.withColumn("key", when(col("key") == "hot_key", rand()).otherwise(col("key")))
```

4.3 Window Functions

Window Specification

```
from pyspark.sql.window import Window
from pyspark.sql.functions import sum as _sum

# Cumulative sum within date range
window_spec = Window \
  .partitionBy("user_id") \
  .orderBy("date") \
  .rowsBetween(Window.unboundedPreceding, Window.currentRow)

df.withColumn("cumulative_spend", _sum("amount").over(window_spec))
```

Performance Considerations

- **Range Frames**: Use .rangeBetween for time-series to avoid row-by-row processing.
- **Partition Sizing**: Too many partitions → Task overhead; too few → Skew.

4.4 Pivot Tables

Dynamic vs Static Pivoting

```
# Dynamic (auto-detects distinct values - risky for high cardinality)
df.groupBy("store").pivot("product").sum("sales")

# Static (controlled columns)
df.groupBy("store").pivot("product", ["apple", "banana"]).sum("sales")
```

Real-World Use Case

Retail analysis: Pivot monthly sales data into columns (Jan_sales, Feb_sales) for trend reporting.

4.5 Handling Missing Data

Advanced Imputation

```
from pyspark.ml.feature import Imputer

imputer = Imputer(
  inputCols=["price"],
  outputCols=["price_imputed"],
  strategy="median"  # "mean", "mode"
).fit(df)
imputer.transform(df)
```

Null Handling in Joins

```
# Treat nulls as valid join keys
spark.conf.set("spark.sql.legacy.nullInPredicateRewrite", True)
```

4.6 User Defined Functions (UDFs)

Pandas UDFs (Vectorized)

```
from pyspark.sql.functions import pandas_udf

@pandas_udf("double")
def pandas_normalize(s: pd.Series) -> pd.Series:
   return (s - s.mean()) / s.std()

df.withColumn("normalized", pandas_normalize(col("value")))
```

When to Avoid UDFs

- **Built-in Alternative**:

```
-- SQL equivalent for categorize_udf
CASE WHEN item_weight > 10 THEN 'Heavy' ELSE 'Light' END
```

UDF Optimization

- **Arrow Serialization**:

```
spark.conf.set("spark.sql.execution.arrow.pyspark.enabled", "true")
```

Best Practices

1. **Join Strategy**: Use explain() to verify join type (broadcast/sort-merge).
2. **Pivot Limits**: Never pivot columns with >10K distinct values.
3. **Window Tuning**: Pair .orderBy() with .rowsBetween() for deterministic results.

Common Errors

- Cannot broadcast table larger than 8GB: Increase autoBroadcastJoinThreshold.
- Specified window frame appears more than once: Reuse window specs.

Real-World Use Case

A telecom company uses:

- Window functions to calculate 30-day rolling usage.
- left_anti joins to identify churned customers.
- Pivots to compare regional service outages.

Section 5: PySpark Performance Tuning and Spark SQL

5.1 Caching and Persistence

Purpose

Store frequently accessed DataFrames in memory/disk to avoid recomputation.

Storage Levels

Level	Description	Use Case
MEMORY_ONLY	Deserialized Java objects in memory	Fast access for small datasets
MEMORY_AND_DISK_SER	Serialized data (spills to disk if OOM)	Large datasets with limited RAM
DISK_ONLY	Data stored only on disk	Rarely accessed DataFrames

Example

```
# Cache for iterative ML training
train_data = spark.read.parquet("train_data").cache()
for i in range(10):
   model = train_model(train_data)  # Reused without re-reading
train_data.unpersist()
```

Best Practices

- **Avoid Over-Caching**: Cache only DataFrames used ≥3 times.
- **Monitor Usage**:

```
storage = spark.sparkContext.statusTracker().getExecutorMemoryStatus
print(storage)
```

Pitfalls

- **Serialization Overhead**: MEMORY_ONLY_SER saves space but adds CPU cost.
- **Stale Data**: Cached DataFrames don't auto-update if source data changes.

5.2 Partitioning Strategies

Repartitioning

```
# Shuffle data to balance partitions
df = df.repartition(200, "country")  # 200 partitions by country
```

- **Optimal Size**:

```
# Calculate partitions
data_size = spark.sql("DESCRIBE DETAIL df").select("sizeInBytes").first()[0]
num_partitions = data_size // (128 * 1024 * 1024)  # 128MB partitions
```

Coalesce

```
# Merge partitions without shuffle (e.g., after filtering)
df = df.filter(col("year") == 2023).coalesce(50)
```

Skew Handling

```
# Salting skewed keys
df = df.withColumn("salted_key", concat(col("key"), lit("_"), (rand() * 100).cast("int")))
df.groupBy("salted_key").count()
```

5.3 Broadcast Variables

Mechanism

Broadcast variables are sent to all worker nodes via efficient BitTorrent-like protocol.

Configuration

```
# Adjust auto-broadcast threshold (default 10MB)
spark.conf.set("spark.sql.autoBroadcastJoinThreshold", 100 * 1024 * 1024)  # 100MB
```

Example

```
# Force broadcast for a medium-sized table
dim_df = spark.read.parquet("dim_table")
fact_df.join(broadcast(dim_df), "id")
```

Pitfalls

- **OOM Errors**: Broadcasting tables >1/4 of executor memory.
- **Network Overhead**: Large broadcasts delay task scheduling.

5.4 Spark SQL Integration

Temp View Management

```
# Global temp view (cross-session)
df.createGlobalTempView("global_sales")
spark.sql("SELECT * FROM global_temp.global_sales")
```

SQL + DataFrame Interop

```
# Mix SQL and DataFrames
sql_df = spark.sql("SELECT * FROM sales WHERE amount > 100")
joined_df = sql_df.join(df_items, "item_id")
```

BI Integration

```
# Expose via JDBC
./bin/beeline -u jdbc:hive2://localhost:10000
```

5.5 Execution Plan Analysis

Plan Interpretation

- **Physical Plan**:
    - *(1) Project: Column selection.
    - *(2) Filter: WHERE clause execution.
    - Exchange: Shuffle operation (bottleneck).

Example

df.filter(col("sales") > 1000).groupBy("region").count().explain(mode="extended")

Output highlights:

- HashAggregate: Indicates groupBy.
- Exchange hashpartitioning(region): Shuffle before aggregation.

Optimization

- **Predicate Pushdown**: Ensure filters are applied at scan stage.
- **Column Pruning**: Read only necessary columns.

5.6 Adaptive Query Execution (AQE)

Key Features

- **Coalesce Shuffle Partitions**:

spark.conf.set("spark.sql.adaptive.coalescePartitions.minPartitionSize", "64MB")

- **Skew Join Handling**: Splits skewed partitions into smaller chunks.
- **Runtime Join Strategy**: Switches to broadcast join if a table is smaller than expected.

Configuration

# Enable all AQE optimizations
spark.sql.adaptive.enabled=true
spark.sql.adaptive.skewJoin.enabled=true
spark.sql.adaptive.localShuffleReader.enabled=true

Real-World Impact

A telecom company reduced ETL job time by 40% using AQE to handle skewed customer usage data.

Best Practices

1. **Caching**: Prefer MEMORY_AND_DISK_SER for DataFrames >1GB.
2. **Partitioning**: Align partition keys with common filter/join columns.
3. **Broadcast**: Use for star schema joins (fact ↔ dimensions).
4. **Execution Plans**: Check for unnecessary shuffles using explain().

Common Errors

- OutOfMemoryError: Caused by caching oversized DataFrames or over-partitioning.
- Broadcast timeout: Increase spark.sql.broadcastTimeout (default 300s).

Section 6: PySpark in Production

6.1 ETL Pipeline Design

Best Practices

- **Modularity**:

```
# Add validation layer
def validate(df):
  return df.withColumn("is_valid", col("amount") > 0) \
      .filter(col("is_valid"))
```

    - Split pipelines into reusable functions for testing/maintenance.

- **Idempotence**:

```
# Overwrite specific partitions safely
df.write.partitionBy("date") \
    .mode("overwrite") \
    .option("replaceWhere", "date = '2024-01-01'")  # Delta Lake only
```

- **Data Quality**:

```
# Use PySpark Expectations (open-source)
from pyspark_expectations import expect
df = df.withColumn("valid_amount", expect.greater_than(col("amount"), 0))
invalid = df.filter(col("valid_amount") == False)
invalid.write.mode("append").parquet("/error_logs")
```

Pitfalls

- **Schema Drift**: Use Delta Lake schema enforcement to block invalid writes.
- **Partial Failures**: Write to temporary directories first, then atomically commit.

6.2 Structured Streaming

Core Concepts

- **Watermarking**:
    - Late data beyond watermark (e.g., 10 mins) is **dropped**.
    - Enables state cleanup (e.g., groupBy with windows).
- **Checkpointing**:
    - Stores offsets and metadata to recover from failures.
    - **Never modify checkpoint locations** between pipeline versions.

Advanced Sinks

```
# Use foreachBatch for custom sinks (e.g., Cassandra)
def write_to_cassandra(batch_df, batch_id):
  batch_df.write.format("cassandra").mode("append").save()

stream.writeStream.foreachBatch(write_to_cassandra).start()
```

Backpressure Handling

```
spark.conf.set("spark.streaming.backpressure.enabled", "true")  # Kafka only
```

6.3 Machine Learning with MLlib

Model Deployment

```
# Save model to S3/DBFS
model.write().overwrite().save("s3://models/rf_v1")

# Load in production
from pyspark.ml import PipelineModel
prod_model = PipelineModel.load("s3://models/rf_v1")
```

Real-Time Inference

- **Structured Streaming**:

```
stream_df = spark.readStream.format("kafka")...
predictions = prod_model.transform(stream_df)
```

- **Serving via REST**: Export to ONNX and serve via Triton Inference Server.

Cost-Efficient Tuning

```
# Use TrainValidationSplit instead of CrossValidator for large datasets
from pyspark.ml.tuning import TrainValidationSplit
tvs = TrainValidationSplit(estimator=pipeline, estimatorParamMaps=param_grid, trainRatio=0.8)
```

6.4 Production Deployment

Cluster Sizing

Parameter	Recommendation
spark.executor.cores	4-5 cores (avoid hyperthreading)
spark.memory.fraction	0.6 (leave room for OS/buffers)

Dynamic Allocation

```
spark.conf.set("spark.dynamicAllocation.maxExecutors", 100)  # Prevent cost overruns
```

Monitoring

- **Spark UI**: Track task skew via Task Time histograms.
- **Prometheus Integration**:

```
spark.conf.set("spark.metrics.conf.*.sink.prometheus.class",
"org.apache.spark.metrics.sink.PrometheusSink")
```

6.5 Delta Lake Integration

Schema Evolution

- **Add Columns**: autoMerge allows new nullable columns.
- **Breaking Changes** (e.g., rename column):

```
ALTER TABLE delta.`/path` SET TBLPROPERTIES ('delta.enableSchemaEnforcement' = 'false')
```

Time Travel

```
# Compare versions for debugging
df_v1 = spark.read.format("delta").option("versionAsOf", 1).load("/path")
df_v2 = spark.read.format("delta").option("versionAsOf", 2).load("/path")
diff = df_v2.exceptAll(df_v1)
```

Optimizations

- **Z-Ordering**:

```
OPTIMIZE delta.`/path` ZORDER BY (user_id, timestamp)  # Speed up user-centric queries
```

6.6 Error Handling

Retry Strategies

```
from tenacity import retry, stop_after_attempt, wait_exponential

@retry(stop=stop_after_attempt(3), wait=wait_exponential(multiplier=1, min=4, max=10))
def write_to_adls(df):
  df.write.format("delta").save("abfss://...")
```

Comprehensive Validation

```
try:
  df = spark.read.format("parquet").load("s3://data/")
except Exception as e:
  if "Path does not exist" in str(e):
    initialize_storage()
  elif "AccessDenied" in str(e):
    refresh_aws_credentials()
  else:
    raise
```

Best Practices

1. **Circuit Breakers**: Halt pipelines after consecutive failures.
2. **Dead Letter Queues**: Route invalid records to dedicated storage for analysis.

Real-World Use Case

A fintech company uses:

- Delta Lake time travel to audit transaction histories.
- Structured Streaming with watermarking for real-time fraud detection.
- MLlib pipelines retrained weekly with automated hyperparameter tuning.

Section 7: Date Functions in PySpark

7.1 Core Date Functions

1. Current Date/Timestamp

Purpose: Capture the current system date and time.

Syntax:

```
from pyspark.sql.functions import current_date, current_timestamp

# Add current date (static for all rows)
df = df.withColumn("current_date", current_date())

# Add current timestamp (with microseconds)
df = df.withColumn("current_timestamp", current_timestamp())
```

Key Differences:

- current_date() returns DateType (e.g., 2024-05-27).
- current_timestamp() returns TimestampType (e.g., 2024-05-27 14:30:45.123456).

---

2. Date Arithmetic

Purpose: Manipulate dates by adding/subtracting days or calculating intervals.

a. Date Addition/Subtraction

```
from pyspark.sql.functions import date_add, date_sub

# Add 7 days to order_date
df = df.withColumn("delivery_date", date_add(col("order_date"), 7)

# Subtract 3 days (two methods)
df = df.withColumn("deadline", date_sub(col("order_date"), 3)
df = df.withColumn("deadline", date_add(col("order_date"), -3)  # Equivalent
```

b. Date Difference

```
from pyspark.sql.functions import datediff, months_between

# Days between delivery and order dates
df = df.withColumn("days_to_deliver", datediff(col("delivery_date"), col("order_date"))

# Months between dates (returns fractional value)
df = df.withColumn("months_diff", months_between(col("end_date"), col("start_date")))
```

---

3. Date Formatting

Purpose: Convert dates to custom string formats or parse strings into dates.

a. Formatting Dates

```
from pyspark.sql.functions import date_format

# Convert DateType to formatted strings
df = df.withColumn("formatted_date",
   date_format(col("order_date"), "dd-MMM-yyyy")  # e.g., 27-May-2024
)

# Common format patterns:
# "yyyy-MM-dd" → 2024-05-27
# "MM/dd/yyyy" → 05/27/2024
# "EEEE" → Monday (full weekday name)
```

b. Parsing Strings to Dates

```
python
Copy
from pyspark.sql.functions import to_date

# Parse string column to DateType
df = df.withColumn("parsed_date",
   to_date(col("date_string"), "dd/MM/yyyy")  # e.g., "27/05/2024" → DateType
)
```

---

4. Date Component Extraction

Purpose: Extract year, month, day, or weekday from dates.

```
from pyspark.sql.functions import year, month, dayofweek

df = df.withColumn("order_year", year(col("order_date")))
df = df.withColumn("order_month", month(col("order_date")))
df = df.withColumn("order_day_of_week", dayofweek(col("order_date")))  # Sunday=1, Saturday=7
```

---

5. Time Zone Handling

Purpose: Convert timestamps across time zones.

```
from pyspark.sql.functions import from_utc_timestamp, to_utc_timestamp
# Convert UTC to local time (e.g., "America/New_York")
df = df.withColumn("local_time",
   from_utc_timestamp(col("utc_timestamp"), "America/New_York")
)
# Convert local time to UTC
df = df.withColumn("utc_time",
   to_utc_timestamp(col("local_timestamp"), "Europe/Paris")
)
```

Best Practices

1. **ISO 8601 Format**: Store dates as yyyy-MM-dd for consistency.
2. **Avoid String Manipulation**: Use built-in functions instead of UDFs for performance.
3. **Time Zone Awareness**: Always store timestamps in UTC and convert to local time zones on read.

Common Pitfalls

1. **Incorrect Format Strings**:

```
# Fails silently if format mismatches
df.withColumn("date", to_date(col("date_str"), "yyyy-MM-dd"))  # Returns null for "27/05/2024"
```

Fix: Validate data or use try_to_date (Spark 3.0+).

2. **Leap Year/Invalid Dates**:
   - to_date("2023-02-30", "yyyy-MM-dd") → null.
     Mitigation: Use data validation pipelines.
3. **Time Zone Ignorance**: Mixing timestamps without converting to UTC.

Real-World Use Case

A logistics company uses these functions to:

1. Calculate delivery SLAs:

```
df.withColumn("sla_met",
  datediff(col("delivery_date"), col("order_date")) <= 3
)
```

2. Generate daily reports with date_format(col("timestamp"), "yyyy-MM-dd").
3. Convert global order timestamps to local time zones for customer notifications.

Advanced Example

```
# Calculate business days (excluding weekends)
from pyspark.sql.functions import when, dayofweek

df = df.withColumn("business_days",
  when(dayofweek(col("order_date")).isin(1, 7),  # Sunday=1, Saturday=7
  date_add(col("order_date"), 2  # Skip weekend
).otherwise(date_add(col("order_date"), 1))
)
```

Section 8: Handling Nulls in PySpark

8.1 Handling Nulls in PySpark

1. Dropping Null Values

Purpose: Remove rows/columns containing nulls to clean datasets.

Syntax & Parameters

```
# Drop rows with ANY nulls in specified columns
df_clean = df.dropna(how="any", subset=["order_date", "customer_id"])

# Drop rows where ALL specified columns are null
df_clean = df.dropna(how="all", subset=["comment", "feedback"])

# Drop rows with ANY nulls in all columns (default behavior)
df_clean = df.dropna()
```

Use Cases

- Critical columns (e.g., order_id, timestamp) where nulls indicate data corruption.
- Aggregation prep: Ensure key grouping columns are non-null.

---

2. Filling Null Values

Purpose: Replace nulls with meaningful defaults or calculated values.

Basic Replacement

```
# Fill all nulls in numeric columns with 0
df_filled = df.fillna(0)

# Column-specific replacements
df_filled = df.fillna(
  {"outlet_size": "Unknown", "item_weight": df.agg(avg("item_weight")).first()[0]}
)
```

Advanced Conditional Filling

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

# Fill "revenue" nulls with 0, "country" nulls with "N/A"
df = df.withColumn("revenue", coalesce(col("revenue"), lit(0)) \
    .withColumn("country", when(col("country").isNull(), "N/A").otherwise(col("country")))
```

---

3. Specialized Null Handling

a. Forward/Backward Filling

```
from pyspark.sql.window import Window
from pyspark.sql.functions import last
# Fill nulls with last known value (e.g., time-series data)
window = Window.orderBy("timestamp").rowsBetween(Window.unboundedPreceding,
Window.currentRow)
df = df.withColumn("filled_value", last("value", ignorenulls=True).over(window))
```

b. Statistical Imputation

```
# Compute median using approxQuantile (for large datasets)
median = df.approxQuantile("item_weight", [0.5], 0.01)[0]
df = df.fillna(median, subset=["item_weight"])
```

---

4. Null Detection & Analysis

Purpose: Audit datasets to quantify null impact.

```
from pyspark.sql.functions import col, sum as spark_sum

# Count nulls per column
null_counts = df.select([
  spark_sum(col(c).isNull().cast("int")).alias(c)
  for c in df.columns
]).collect()[0].asDict()

# Output: {"order_date": 12, "customer_id": 0, ...}
```

---

5. Best Practices

1. **Avoid Over-Dropping**:

```
# Only drop nulls in critical columns, retain others for analysis
df.dropna(subset=["order_id"])
```

2. **Documentation**:
    - Log null counts before/after handling.
    - Track replacement logic (e.g., "Filled outlet_size nulls with 'Unknown'").
3. **Validation**:

```
# Ensure no nulls remain in key columns
assert df.filter(col("order_id").isNull()).count() == 0
```

---

6. Real-World Scenario

Problem:

- Outlet_Size has 15% nulls (fill with "Unknown").
- Order_Date has 0.1% nulls (drop these rows).

Solution:

```
df = (df
  .fillna("Unknown", subset=["Outlet_Size"])
  .dropna(subset=["Order_Date"])
)
```

---

7. Common Pitfalls

1. **Silent Data Loss**:
    - Using how="any" without subset drops rows with nulls in unimportant columns.
2. **Inappropriate Fill Values**:
    - Filling numeric nulls with 0 might skew aggregations (use median instead).
3. **Cascade Effects**:

- Aggregations like groupBy automatically exclude nulls, leading to undercounts.

8. Advanced Techniques

Custom Null Markers

```
# Treat "N/A" and "?" as nulls during read
df = spark.read.csv("data.csv", nullValue=["N/A", "?"])
```

Nullable Schema Control

```
from pyspark.sql.types import StructType, StructField, StringType

# Enforce non-nullable schema
schema = StructType([
  StructField("order_id", StringType(), nullable=False)
])
df = spark.read.schema(schema).csv("orders.csv")
```

Section 9: Split and Explode Functions

9.1 Split and Explode Functions in PySpark

1. Split Function

Purpose: Convert delimited strings into arrays for structured processing.

Basic Syntax

```
from pyspark.sql.functions import split

# Split a string column by delimiter
df = df.withColumn("type_components", split(col("Outlet_Type"), " "))
```

Input/Output:

- "Supermarket Type1" → ["Supermarket", "Type1"] (ArrayType(StringType))

Advanced Usage

```
# Split with regex (e.g., split by multiple delimiters)
df = df.withColumn("tags", split(col("description"), "[ ,;]+"))

# Limit splits (only first 2 occurrences)
df = df.withColumn("limited_split", split(col("path"), "/", 2))
```

2. Array Indexing

Purpose: Access specific elements without exploding the entire array.

getItem() Method

```
# Extract first and second array elements
df = df.withColumn("main_type", col("type_components").getItem(0)) \
    .withColumn("sub_type", col("type_components").getItem(1))
```

Performance Note:

- 10-100x faster than explode() when only specific elements are needed.

Negative Indexing

```
# Get last element in array
df = df.withColumn("last_tag", col("tags").getItem(-1))
```

---

3. Explode Function

Purpose: Transform array elements into individual rows (denormalization).

Basic Explode

```
from pyspark.sql.functions import explode

# Each array element becomes a new row
df_exploded = df.withColumn("tag", explode(col("tags")))
```

Before/After:

id	tags	→	id	tag
1	[A, B]	→	1	A
		→	1	B

Explode with Posexplode

```
from pyspark.sql.functions import posexplode

# Include array index (position)
df = df.withColumn("pos_tag", posexplode(col("tags"))) \
    .select("id", "pos_tag.pos", "pos_tag.col")
```

Output:

id	pos	col
1	0	A
1	1	B

---

4. Real-World Use Cases

a. Normalizing CSV-like Columns

```
# Input: "apple,banana,orange" → Output: 3 rows
df = df.withColumn("fruit", explode(split(col("fruits_csv"), ",")))
```

b. Processing Nested JSON Arrays

```
# JSON input: {"sizes": ["S", "M", "L"]}
df = df.withColumn("size", explode(col("sizes")))
```

c. Tag Analysis

```
# Count occurrences of each tag
df_tags = df.withColumn("tag", explode(col("tags"))) \
        .groupBy("tag").count()
```

5. Performance Considerations

Explosion Factor

- Exploding an array with average length *N* increases row count by *Nx*.
- **Mitigation**:

```
# Filter arrays before exploding
df = df.withColumn("large_tags", filter(col("tags"), lambda x: length(x) > 3))
```

Alternatives to Explode

- Use array_contains() for existence checks:

```
df.filter(array_contains(col("tags"), "urgent"))
```

- Use transform() for in-array processing:

```
df.withColumn("upper_tags", expr("transform(tags, x -> upper(x))"))
```

---

6. Best Practices

1. **Schema Control**:

```
# Ensure exploded columns are typed correctly
df.withColumn("size", explode(col("sizes")).cast(StringType()))
```

2. **Error Handling**:

```
# Skip null arrays
df.withColumn("tag", explode(coalesce(col("tags"), array())))
```

3. **Explode Last**: Apply after filtering to minimize data volume.

---

7. Common Pitfalls

- **Cartesian Explosion**: Exploding multiple array columns without caution:

```
# DANGER: Rows multiply by len(tags)*len(sizes)
df.withColumn("tag", explode("tags")) \
  .withColumn("size", explode("sizes"))
```

Fix: Use arrays_zip() for correlated explosions:

```
df.withColumn("tag_size", explode(arrays_zip("tags", "sizes")))
```

- **Null Propagation**: Exploding null arrays yields zero rows (silent data loss).

Section 10: Array Operations (array_contains, collect_list)

10.1 Array Operations in PySpark

1. array_contains()

Purpose: Check for the existence of a value in an array column.

Syntax & Usage

```
from pyspark.sql.functions import array_contains

# Check if "Type1" exists in the array column
df = df.withColumn("is_type1", array_contains(col("split_col"), "Type1"))
```

Output:

split_col	is_type1
["Supermarket", "Type1"]	true
["Grocery", "Type2"]	false

Use Cases:

- Filtering records with specific tags:

```
df.filter(array_contains(col("tags"), "urgent")).show()
```

- Feature engineering for ML (e.g., flagging products with certain attributes).

---

2. collect_list()

Purpose: Aggregate values into an array **with duplicates**, preserving order.

Syntax & Example

```
from pyspark.sql.functions import collect_list

# Aggregate all user actions into an array
df_actions = df.groupBy("user_id") \
        .agg(collect_list("action").alias("action_sequence"))
```

Output:

user_id	action_sequence
1001	["login", "view", "view", "buy"]

Key Features:

- Retains duplicates and order (e.g., chronological user activity logs).
- Returns ArrayType matching the input column's data type.

---

3. collect_set()

Purpose: Aggregate values into an array **without duplicates**.

Syntax & Example

```
from pyspark.sql.functions import collect_set

# Aggregate unique products viewed per user
df_unique = df.groupBy("user_id") \
        .agg(collect_set("product_id").alias("unique_products"))
```

Output:

user_id	unique_products
1001	["P123", "P456"]

Performance Note:

- Deduplication requires additional computation (hash + sort).
- Avoid on high-cardinality columns (e.g., timestamps).

---

4. Comparison: collect_list **vs** collect_set

Feature	collect_list	collect_set
Duplicates	Retained	Removed
Order	Preserved	Not guaranteed
Performance	Faster	Slower (due to dedup)
Use Case	Action sequences	Unique item aggregation

---

5. Advanced Use Cases

a. Nested Aggregations

```
# Collect arrays of structs
df.groupBy("department") \
  .agg(collect_list(struct(col("employee_id"), col("salary"))).alias("employees"))
```

b. Size Analysis

```
from pyspark.sql.functions import size

# Count items in collected arrays
df_actions.withColumn("action_count", size(col("action_sequence")))
```

c. Explode + Collect

```
# Normalize and re-aggregate (e.g., cleaning tags)
df_exploded = df.withColumn("tag", explode(col("tags"))) \
        .filter(length(col("tag")) > 3)
df_clean = df_exploded.groupBy("id").agg(collect_set("tag"))
```

---

6. Best Practices

1. **Use** collect_list **for Ordered Data**:
    - User session streams, time-series event sequences.
2. **Use** collect_set **for Uniqueness**:
    - Distinct product views, unique IP addresses.
3. **Limit Array Sizes**:

```
# Truncate arrays to avoid OOM errors
df.withColumn("trimmed_actions", slice(col("action_sequence"), 1, 100))
```

---

7. Common Pitfalls

- **Memory Overload**: Collecting large arrays (e.g., 1M elements) → Driver OOM.
    - **Fix**: Use agg(collect_list(...).alias(...)) only if necessary for downstream processing.
- **Order Non-Guarantee in** collect_set:

```
# Use collect_list + array_distinct for order preservation
df.groupBy("user_id") \
  .agg(array_distinct(collect_list("product_id")).alias("unique_ordered"))
```

---

8. Real-World Applications

a. Recommendation Systems

```
# Collect co-viewed products
df_co_view = df.groupBy("session_id") \
        .agg(collect_set("product_id").alias("related_products"))
```

b. Data Quality Checks

```
# Find records missing required tags
df.filter(~array_contains(col("tags"), "validated")) \
  .write.parquet("/path/invalid_records")
```

c. User Behavior Analysis

```
# Track full clickstream vs unique page visits
df.groupBy("user_id") \
  .agg(
    collect_list("page_url").alias("clickstream"),
    collect_set("page_url").alias("unique_pages")
  )
```

Section 11: GroupBy and Aggregations in PySpark

1. Basic GroupBy Syntax

Concept: Grouping data by one or more columns and applying aggregate functions.

Syntax:

```
from pyspark.sql.functions import sum

aggregated_df = df.groupBy("Category") \
        .agg(sum("Sales").alias("total_sales"))
```

Example:

```
# Sample DataFrame
data = [("Electronics", 100), ("Clothing", 200), ("Electronics", 300)]
df = spark.createDataFrame(data, ["Category", "Sales"])

# GroupBy + Sum
result = df.groupBy("Category").agg(sum("Sales").alias("total_sales"))
result.show()
```

Category	Total Sales
Electronics	400
Clothing	200

Key Notes:

- Use groupBy() to specify grouping columns.
- agg() accepts aggregate expressions (e.g., sum(), avg()).

---

2. Common Aggregation Functions

Core Functions:

Function	Description	Example Usage
sum("col")	Sum of numeric values	sum("Sales")
avg("col")	Average value (alias: mean())	avg("Price")
count("col")	Non-null count	count("OrderID")
min("col")/max()	Minimum/maximum value	min("Temperature")
collect_list("col")	Collect values into a list (duplicates allowed)	collect_list("Product")
collect_set("col")	Collect unique values into a list	collect_set("Customer")

Example:
```
from pyspark.sql.functions import avg, collect_list

df.groupBy("Category") \
  .agg(avg("Sales").alias("avg_sales"),
     collect_list("Sales").alias("all_sales")) \
  .show()
```

Category	Avg Sales	All Sales
Electronics	200.0	[100, 300]
Clothing	200.0	[200]

3. Multiple Aggregations
Concept: Apply multiple aggregate functions in a single groupBy operation.
Example:
```
from pyspark.sql.functions import sum, avg, count

df.groupBy("Department") \
  .agg(sum("Sales").alias("total_sales"),
     avg("Sales").alias("avg_sales"),
     count("*").alias("transactions")) \
  .show()
```

Best Practices:

- Use * in count("*") to include rows with null values.
- Always **alias columns** for readability (e.g., alias("total_sales")).

4. Pivot Operations
Concept: Reshape data by converting unique values of a column into multiple columns.
Syntax:
```
pivoted_df = df.groupBy("Year").pivot("Quarter").sum("Revenue")
```

Example:
```
python
Copy
# Sample Data
data = [("2023", "Q1", 500), ("2023", "Q2", 600), ("2024", "Q1", 700)]
df = spark.createDataFrame(data, ["Year", "Quarter", "Revenue"])

# Pivot by Quarter
pivoted = df.groupBy("Year").pivot("Quarter").sum("Revenue")
pivoted.show()
```

Year	Q1	Q2	Q3
2023	500	600	
2024	700		

Advanced Use Case:
# Multiple Aggregations with Pivot
df.groupBy("Year").pivot("Quarter").agg(sum("Revenue"), avg("Revenue"))

---

5. Performance Considerations

1. **Use Approximate Counts**:
    - Replace countDistinct() with approx_count_distinct() for large datasets (faster but slightly inaccurate):

```
from pyspark.sql.functions import approx_count_distinct
df.agg(approx_count_distinct("CustomerID", rsd=0.05))
```

2. **Filter Early**:
    - Reduce data size before grouping:

```
df.where(col("Sales") > 100).groupBy("Category").sum()
```

3. **Cache Aggregated Data**:
    - Persist reused DataFrames:

```
aggregated_df = df.groupBy(...).agg(...).cache()
```

---

6. Real-World Example
Scenario: Analyze sales data for a retail chain.
Code:

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

sales_report = (df.groupBy("store_id", "product_category")
        .agg(sum("sales_amount").alias("total_sales"),
           avg("unit_price").alias("avg_price"),
           countDistinct("customer_id").alias("unique_customers"))
        .orderBy("total_sales", ascending=False))
```

Output Columns:

- total_sales: Total revenue per category per store.
- avg_price: Average product price.
- unique_customers: Distinct customers (using countDistinct).

---

7. Common Pitfalls & Solutions

1. **Unaliased Columns**:
    - **Issue**: Aggregation columns get ugly names like sum(Sales).
    - **Fix**: Always use .alias("readable_name").
2. **Data Skew**:
    - **Issue**: Grouping by high-cardinality columns (e.g., user_id) causes uneven partitions.
    - **Fix**: Use salting or pre-aggregation.
3. **Null Handling**:
    - **Issue**: avg() ignores nulls, but count("col") does not.
    - **Fix**: Clean data with fillna() or coalesce() before aggregation:

```
df.fillna(0, subset=["Sales"])
```

Summary

- **GroupBy + Agg** is essential for summarizing data (e.g., sales reports, KPIs).
- Use **pivot** to reshape data for visualization or reporting.
- Optimize with **filtering**, **approximate functions**, and **caching** for large datasets.

Section 12: Window Functions in PySpark

1. Core Concept

What Are Window Functions?

Window functions perform calculations across a set of rows related to the current row *without collapsing the dataset*. Unlike groupBy, they retain all original data while adding computed columns (e.g., rankings, running totals).

Key Characteristics:

- Preserve row-level details.
- Define a "window" of rows using partitions, ordering, and frame boundaries.
- Commonly used for time-series analysis, rankings, and cumulative metrics.

---

2. Key Window Functions

Function	Description	Example Use Case
row_number()	Assigns a unique sequential number to rows within a partition.	Ranking salespeople with no ties.
rank()	Assigns ranks with gaps for ties (e.g., 1, 2, 2, 4).	Ranking students by test scores.
dense_rank()	Assigns ranks without gaps for ties (e.g., 1, 2, 2, 3).	Competition standings.
lag(col, offset)	Returns the value from a previous row (offset = 1 for immediate previous).	Comparing current month's sales to prior.
lead(col, offset)	Returns the value from a subsequent row.	Forecasting next month's sales.

Example:

```
from pyspark.sql import Window
from pyspark.sql.functions import row_number, rank, dense_rank, lag

# Sample DataFrame
data = [("Sales", 1000), ("Sales", 1500), ("HR", 800), ("Sales", 1200)]
df = spark.createDataFrame(data, ["department", "salary"])

# Define window specification
window_spec = Window.partitionBy("department").orderBy(col("salary").desc())

# Apply window functions
result = df.withColumn("row_number", row_number().over(window_spec)) \
        .withColumn("rank", rank().over(window_spec)) \
        .withColumn("dense_rank", dense_rank().over(window_spec)) \
        .withColumn("prev_salary", lag("salary", 1).over(window_spec))

result.show()
```

Department	Salary	Row Number	Rank	Dense Rank	Prev Salary
Sales	1500	1	1	1	null
Sales	1200	2	2	2	1500
Sales	1000	3	3	3	1200
HR	800	1	1	1	null

3. Window Specification

Syntax:

```
from pyspark.sql.window import Window

window_spec = Window \
   .partitionBy("partition_col") \    # Split data into groups
   .orderBy("order_col") \           # Sort rows within partitions
   .rowsBetween(start, end)          # Define frame boundaries
```

Frame Boundaries:

- rowsBetween(start, end): Physical row offsets
  (e.g., Window.unboundedPreceding, 1, Window.currentRow).
- rangeBetween(start, end): Logical value offsets (e.g., for dates or numeric ranges).

Examples:

1. **Cumulative Sum (Entire Partition)**:

Window.partitionBy("department").orderBy("date").rowsBetween(Window.unboundedPreceding, Window.currentRow)

2. **3-Day Rolling Average**:

Window.partitionBy("store_id").orderBy("date").rangeBetween(-2, 0) # 3-day window

---

4. Practical Examples

a) Running Totals

```
from pyspark.sql.functions import sum

window_spec =
Window.partitionBy("department").orderBy("date").rowsBetween(Window.unboundedPreceding,
Window.currentRow)
df.withColumn("running_total", sum("sales").over(window_spec)).show()
```

Department	Date	Sales	Running Total
Sales	Jan 1	100	100
Sales	Jan 2	200	300
HR	Jan 1	50	50

b) Top 3 Employees per Department

```
from pyspark.sql.functions import row_number

window_spec = Window.partitionBy("department").orderBy(col("salary").desc())
ranked_df = df.withColumn("rank", row_number().over(window_spec))
ranked_df.filter(col("rank") <= 3).show()
```

c) Month-over-Month Growth

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

window_spec = Window.partitionBy("department").orderBy("month")
df.withColumn("prev_month_sales", lag("sales", 1).over(window_spec)) \
  .withColumn("growth", when(col("prev_month_sales").isNull(), 0)
                .otherwise((col("sales") - col("prev_month_sales")) / col("prev_month_sales")))
```

---

5. Performance Optimization

1. **Narrow Partitions**:
   Avoid partitioning by high-cardinality columns (e.g., user_id). Use partitionBy on columns with limited unique values.
2. **Frame Selection**:
   - Use rangeBetween for numeric/temporal data (e.g., time-series intervals).
   - Use rowsBetween for fixed-size windows (e.g., last 7 rows).
3. **Caching**:
   Cache DataFrames if reusing the same window spec multiple times:

```
df.cache().groupBy(...)  # Reduces re-computation overhead
```

---

6. Common Pitfalls & Fixes

1. **Missing** orderBy():
   - **Issue**: Functions like lag() or cumulative sums require ordered windows.
   - **Fix**: Always include orderBy in the window spec for ordered operations.
2. **Incorrect Frame Boundaries**:
   - **Issue**: The default frame (when using orderBy) is rangeBetween(Window.unboundedPreceding, Window.currentRow), which may include unintended rows.
   - **Fix**: Explicitly define frames (e.g., rowsBetween(-2, 0) for a 3-row window).
3. **Data Skew**:
   - **Issue**: Partitioning on skewed columns (e.g., 90% data in one partition).
   - **Fix**: Salting or pre-filtering skewed data.

Section 13: Joins in PySpark

1. Join Types

Concept: Combine data from two DataFrames based on a common key.

Types:

Join Type	Behavior	SQL Equivalent	Use Case
Inner	Returns only rows with matching keys in both DataFrames.	INNER JOIN	Intersection of data.
Left	Returns all rows from the left DataFrame + matched rows from the right.	LEFT OUTER JOIN	Enriching data without losing left rows.
Right	Returns all rows from the right DataFrame + matched rows from the left.	RIGHT OUTER JOIN	Rarely used (prefer left join).
Full (Outer)	Returns all rows from both DataFrames.	FULL OUTER JOIN	Merging datasets with partial overlaps.
Anti	Returns rows from the left DataFrame that **do not exist** in the right.	NOT IN / NOT EXISTS	Finding missing records (e.g., no orders).
Semi	Returns left rows that exist in the right (no columns from the right).	EXISTS	Filtering without merging data.

Example:

```
# Sample DataFrames
employees = spark.createDataFrame([(1, "Alice", "IT"), (2, "Bob", "HR")], ["id", "name", "dept"])
departments = spark.createDataFrame([("IT", "Infra"), ("HR", "People")], ["dept", "team"])

# Inner Join
employees.join(departments, "dept", "inner").show()
```

Dept	ID	Name	Team
IT	1	Alice	Infra
HR	2	Bob	People

---

2. Join Syntax

Basic Join:

```
df1.join(df2, df1.key == df2.key, "inner")
```

Handling Duplicate Column Names:

```
# Rename conflicting columns
df2_renamed = df2.withColumnRenamed("key", "key2")
df1.join(df2_renamed, df1.key == df2_renamed.key2, "left")
```

Multiple Conditions:

```
df1.join(df2,
    (df1.key1 == df2.key1) &
    (df1.key2 == df2.key2),
    "inner")
```

---

3. Performance Considerations

a) Broadcast Join:

Use when one DataFrame is small (fits in executor memory). PySpark automatically uses broadcast for tables <10MB (configurable).

```
from pyspark.sql.functions import broadcast

# Explicit broadcast hint
small_df = spark.read.parquet("small_data.parquet")
large_df.join(broadcast(small_df), "key")
```

b) Join Hints:
Override Spark's optimizer for specific strategies:

```
# Sort-merge join (large datasets)
large_df1.join(large_df2.hint("merge"), "key")

# Shuffle hash join
large_df1.join(large_df2.hint("shuffle_hash"), "key")
```

c) Optimization Tips:

1. **Filter Early**: Reduce data size before joining.

```
df1.filter(df1.date > "2023-01-01").join(df2, "key")
```

2. **Avoid Cartesian Products**: Use crossJoin cautiously (computationally expensive).
3. **Monitor Skew**: Use .explain() to check join plan and partition distribution.

---

4. Practical Examples

a) Enriching Customer Data:

```
# Left join to retain all customers, even without orders
customers.join(orders, "customer_id", "left") \
    .select("customer_id", "name", "order_amount")
```

b) Finding Unsold Products (Anti-Join):

```
products.join(sales, "product_id", "anti") \
    .show()
```

Product ID	Product Name
105	Unicorn Toy

c) Complex Condition Join:

```
# Join on multiple columns + additional logic
employees.join(departments,
        (employees.dept_id == departments.id) &
        (employees.location == departments.location))
```

---

5. Common Issues & Solutions

Issue	Solution
Duplicate Columns	Rename columns or use .select() to trim unwanted columns post-join.
Data Skew	Use salting (add a random prefix to keys) or broadcast small tables.
Out-of-Memory Errors	Increase shuffle partitions: spark.conf.set("spark.sql.shuffle.partitions", 200)
Slow Joins	Check for skewed partitions with .groupBy("key").count() and rebalance.

6. Anti-Join Use Case: Identify Inactive Customers

```
active_customers = orders.select("customer_id").distinct()
inactive_customers = customers.join(active_customers, "customer_id", "anti")
inactive_customers.show()
```

Summary

- **Join Types**: Choose based on whether you need unmatched rows (left/anti) or full merges (inner/full).
- **Performance**: Use broadcast for small tables, filter early, and monitor skew.
- **Best Practice**: Always alias/rename conflicting columns before joining.

Section 14: Pivot Operations in PySpark

1. Core Concept

What is Pivoting?

Pivoting reshapes data by converting unique values from a single column into multiple columns, enabling cross-tabular summaries (e.g., quarterly sales per row → columns for Q1, Q2, etc.).

Use Cases:

- Financial reporting (quarterly/yearly summaries).
- Survey data analysis (questions as columns).
- Time-series reshaping (hourly/daily metrics).

---

2. Basic Pivot Syntax

Syntax:

```
pivoted_df = df.groupBy("group_cols").pivot("pivot_col").agg(agg_function)
```

Example:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

# Sample DataFrame
data = [
  ("2023", "Q1", 1500000),
  ("2023", "Q2", 1800000),
  ("2024", "Q1", 1600000),
  ("2023", "Q3", 2200000)
]
df = spark.createDataFrame(data, ["Year", "Quarter", "Revenue"])

# Pivot on "Quarter" with sum aggregation
pivoted = df.groupBy("Year").pivot("Quarter").sum("Revenue")
pivoted.show()
```

Year	Q1	Q2	Q3	Q4
2023	1500000	1800000	2200000	null
2024	1600000	null	null	null

Notes:

- Unpivoted rows (e.g., Q4 in 2023) become null by default.
- Use .na.fill(0) to replace nulls with zeros.

3. Advanced Pivot Options

a) Multiple Aggregations:

Apply multiple aggregate functions to the pivoted columns.

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

# Pivot with sum and avg
df.groupBy("Department") \
  .pivot("Product_Category") \
  .agg(sum("Sales").alias("total_sales"),
     avg("Price").alias("avg_price")) \
  .show()
```

Department	Electronics Sum	Electronics Avg	Clothing Sum
IT	50000	250	30000

b) Static Pivoting (Explicit Values):

Manually specify pivot values for better performance.

```
# Define allowed quarters to avoid scanning all data
df.groupBy("Year") \
  .pivot("Quarter", ["Q1", "Q2", "Q3", "Q4"])  # Explicit list
  .sum("Revenue")
```

4. Dynamic vs. Static Pivots

Type	Description	Performance
Dynamic	Automatically detects all distinct values of the pivot column.	Slower (scans data to find values).
Static	Uses a predefined list of pivot values.	Faster (avoids value detection).

When to Use Static:

- Fixed categories (e.g., months, quarters).
- Known pivot values (e.g., survey questions Q1–Q10).

Performance Tip:
Static pivots can be **2–3x faster** than dynamic pivots for large datasets.

---

5. Handling Nulls in Pivoted Data
Replace nulls after pivoting using na.fill():

```
pivoted = df.groupBy("Region").pivot("Product").sum("Sales")
pivoted.na.fill(0).show()

# Before fill:
```

Region	Widget	Gadget
East	1000	null

After fill:

Region	Widget	Gadget
East	1000	0

---

6. Real-World Use Cases
a) Sales Report by Quarter:

```
sales.groupBy("Salesperson") \
    .pivot("Quarter", ["Q1", "Q2", "Q3", "Q4"]) \
    .agg(sum("Amount").alias("Total_Sales"),
         count("*").alias("Transactions")) \
    .na.fill(0)
```

b) Survey Response Analysis:

```
# Each row: Respondent ID, Question ID, Response
surveys.groupBy("Respondent_ID") \
      .pivot("Question_ID", ["Q1", "Q2", "Q3"]) \
      .agg(first("Response"))  # Assumes one response per respondent
```

c) Hourly Sensor Data Reshaping:

```
sensor_data.groupBy("Device_ID") \
         .pivot("Hour", [f"{h:02d}" for h in range(24)])  # "00", "01", ..., "23"
         .avg("Reading") \
         .na.fill(-1)  # Indicate missing readings
```

---

7. Performance Optimization

1. **Filter Early**:
   Reduce data size before pivoting:

```
df.filter(df.Year == 2023).groupBy(...).pivot(...)
```

2. **Use Static Values**:
   Predefine pivot values to avoid dynamic detection:

```
.pivot("Month", ["Jan", "Feb", ..., "Dec"])
```

3. **Tune Shuffle Partitions**:
   Increase partitions for large pivots to avoid OOM errors:

```
spark.conf.set("spark.sql.shuffle.partitions", 200)
```

8. Common Pitfalls & Solutions

Pitfall	Solution
High-Cardinality Pivot	Avoid pivoting on columns with >10K unique values (e.g., user IDs). Use groupBy + collect_list instead.
Nulls in Output	Use .na.fill() post-pivot.
Slow Performance	Use static pivots, filter data, or increase partitions.

Summary

- **Pivoting** reshapes data for summaries and reports.
- **Static Pivots** outperform dynamic ones by avoiding value detection.
- **Optimize** with filtering, predefined values, and null handling.

Section 15: User-Defined Functions (UDFs) in PySpark

1. Core Concept

What Are UDFs?

User-Defined Functions (UDFs) allow you to apply custom Python logic to PySpark DataFrames. While powerful, they bypass Spark's built-in optimizations and introduce performance overhead due to data serialization between the JVM and Python.

When to Use UDFs:

- Complex transformations not achievable with native Spark functions (e.g., regex patterns, custom math operations).
- Prototyping before migrating to Scala UDFs for production.
- Integrating third-party Python libraries (e.g., NLP, geospatial tools).

2. UDF Lifecycle

Step 1: Define the Python Function

```
def reverse_string(s: str) -> str:
   return s[::-1] if s else None
```

Step 2: Register as a Spark UDF

Specify the return type to help Spark manage data serialization:

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

reverse_udf = udf(reverse_string, StringType())  # Return type must match the function's output
```

Step 3: Apply to DataFrame

```
df = spark.createDataFrame([("hello",), ("world",)], ["text"])
df.withColumn("reversed_text", reverse_udf(col("text"))).show()

# Output:
```

Text	Reversed Text
hello	olleh
world	dlrow

---

3. Performance Considerations

Key Limitations:

- **Serialization Overhead**: Data must be serialized between JVM (Spark) and Python, adding latency.
- **No Catalyst Optimization**: Spark's query optimizer cannot optimize UDF logic.
- **Speed**: UDFs are **2–10x slower** than native Spark functions.

When to Avoid UDFs:

- Simple operations achievable with pyspark.sql.functions (e.g., upper(), substring()).
- Large-scale data transformations (use Spark SQL or Scala UDFs instead).

---

4. Advanced UDF Patterns

a) Pandas UDFs (Vectorized UDFs)

Process data in batches using Pandas for near-native performance:

```
from pyspark.sql.functions import pandas_udf
import pandas as pd

@pandas_udf(IntegerType())
def vectorized_square(s: pd.Series) -> pd.Series:
   return s ** 2  # Operate on entire Series instead of row-by-row

df.withColumn("squared", vectorized_square(col("value")))
```

Why Faster?:

- Processes data in chunks (Arrow-based serialization).
- Leverages Pandas' vectorized operations.

b) Complex Return Types

Return structured data using StructType:

```
from pyspark.sql.types import StructType, StructField, BooleanType

schema = StructType([
  StructField("is_valid", BooleanType()),
  StructField("normalized", StringType())
])

@udf(schema)
def validate_email(email: str):
  import re
  pattern = r"^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$"
  is_valid = bool(re.match(pattern, email))
  normalized = email.strip().lower() if is_valid else None
  return (is_valid, normalized)

df.withColumn("email_status", validate_email(col("email"))) \
 .select("email", "email_status.*") \
 .show()
```

Email	Is Valid	Normalized
John@example.com	true	john@example.com

5. Optimization Strategies

1. **Prefer Built-in Functions**:
   Use native Spark operations (e.g., regexp_extract() instead of custom regex UDFs).

2. **Batch Processing with Pandas UDFs**:
   For ML inference or statistical operations:

```
@pandas_udf(ArrayType(FloatType()))
def batch_predict(data: pd.Series) -> pd.Series:
  model = load_ml_model()
  return pd.Series([model.predict(x) for x in data])
```

3. **Cache Before UDFs**:
   Reduce recomputation overhead:

```
df.cache().withColumn("udf_result", slow_udf(col("data")))
```

4. **Tune Partitions**:
   Balance workload across executors:

```
spark.conf.set("spark.sql.shuffle.partitions", 200)
```

---

6. Real-World Examples

a) Address Standardization:

```
@udf(StringType())
def clean_address(address):
  import re
  return re.sub(r'\s+', ' ', address).strip().upper()

df.withColumn("clean_addr", clean_address(col("raw_address")))
```

b) DNA Sequence Analysis:

```
@pandas_udf(IntegerType())
def count_gc_content(sequences: pd.Series) -> pd.Series:
  return sequences.apply(lambda s: s.upper().count("G") + s.upper().count("C"))

df.withColumn("gc_count", count_gc_content(col("dna_sequence")))
```

---

7. Debugging Tips

1. **Test Python Code Locally**:
   Validate logic outside Spark:

```
assert reverse_string("hello") == "olleh"
```

2. **Check Schema Compatibility**:
   Use .printSchema() to verify UDF output types.

3. **Monitor Spark UI**:
   Identify UDF-related bottlenecks in the "SQL" tab (look for PythonRDD tasks).

---

8. Production Considerations

Scala UDFs:

For mission-critical pipelines, rewrite UDFs in Scala:

```
spark.udf.register("scala_udf", (s: String) => s.reverse)
```

Benefits:

- Avoid Python-JVM serialization.
- Full Catalyst optimization.

Summary

- **UDFs** enable custom logic but incur performance costs.
- **Pandas UDFs** bridge the gap with vectorized operations.
- **Avoid** UDFs for simple tasks—always prioritize native Spark functions.

Section 16: Writing Data in PySpark

1. Output Modes

Concept: Control how data is written to storage when the target path/table already exists.

Mode	Behavior	Use Case
append	Adds new rows to existing data.	Streaming pipelines, incremental updates (e.g., daily logs).
overwrite	Deletes existing data and replaces it entirely.	Batch full-refresh (e.g., nightly ETL).
ignore	No operation if data/table already exists.	Conditional writes (e.g., first-time setup).
errorIfExists	Throws an error if the target path/table exists (default behavior).	Preventing accidental data loss.

Example:

```
# Append to existing Parquet dataset
df.write.mode("append").parquet("/data/sales")

# Overwrite CSV with headers
df.write.mode("overwrite").option("header", True).csv("/data/reports")
```

2. File Formats

a) Columnar Formats (Optimized for Analytics):

- **Parquet** (Default):

```
df.write.parquet("/data/output/parquet")  # Snappy compression by default
```

  - **Benefits**: Columnar storage, predicate pushdown, efficient compression.
  - **Use Case**: OLAP workloads, large-scale analytics.

- **Delta Lake**:

```
df.write.format("delta").save("/data/output/delta")  # Requires Delta Lake library
```

  - **Benefits**: ACID transactions, time travel, schema evolution.
  - **Use Case**: Data lakes requiring versioning/audit trails.

b) Row-Based Formats:

- **CSV**:

```
df.write.option("header", True).csv("/data/output/csv")  # Add delimiter, escape chars
```

  - **Use Case**: Interoperability with Excel/legacy systems.

- **JSON**:

df.write.json("/data/output/json") # Line-delimited JSON

  - **Use Case**: Semi-structured data pipelines.

Performance Comparison:

Format	Compression	Schema Handling	Read Speed	Write Speed
Parquet	Excellent	Embedded	Fastest	Fast
Delta	Excellent	Embedded + History	Fast	Moderate
ORC	Good	Embedded	Fast	Fast
CSV	None	External (manual)	Slowest	Slow
JSON	None	Inferred on read	Slow	Moderate

---

3. Partitioning Strategies

Concept: Organize data into directories for faster query performance.

Basic Partitioning:

df.write.partitionBy("year", "month").parquet("/data/partitioned")

- **Directory Structure**:

Copy

/data/partitioned/year=2023/month=01/...

/data/partitioned/year=2023/month=02/...

Advanced Partitioning:

```
# Bucketing + Sorting (Hive Metastore required)
df.write \
  .bucketBy(50, "customer_id") \        # 50 buckets per partition
  .sortBy("transaction_date") \         # Sort within buckets
  .saveAsTable("sales_data")            # Managed table
```

Guidelines:

1. **Partition Size**: Aim for 100MB–1GB per partition.
2. **Avoid Over-Partitioning**: >10K partitions can degrade metadata performance.
3. **Pre-Filter**: Use .filter() before writing to skip unnecessary data.

---

4. Table Management

a) Managed Tables:

- Spark controls *both data and metadata*.
- Dropping the table deletes the data.

df.write.saveAsTable("managed_sales") # Stores data in Spark SQL warehouse

b) External Tables:

- Spark manages metadata only; data resides in external storage.
- Dropping the table only removes metadata.

```
df.write \
  .option("path", "/external/data/sales").saveAsTable("external_sales")
```

5. Cloud-Specific Optimizations

AWS S3:

```
# IAM role-based authentication
spark.conf.set("spark.hadoop.fs.s3a.aws.credentials.provider",
        "com.amazonaws.auth.InstanceProfileCredentialsProvider")

# Write to S3 with server-side encryption
df.write.option("fs.s3a.server-side-encryption-algorithm", "AES256") \
  .parquet("s3a://bucket/data")
```

Azure ADLS Gen2:

```
# Service principal authentication
spark.conf.set("fs.azure.account.auth.type.<storage>.dfs.core.windows.net", "OAuth")
spark.conf.set("fs.azure.account.oauth.provider.type.<storage>.dfs.core.windows.net",
        "org.apache.hadoop.fs.azurebfs.oauth2.ClientCredsTokenProvider")
spark.conf.set("fs.azure.account.oauth2.client.id.<storage>.dfs.core.windows.net", "<client-id>")
spark.conf.set("fs.azure.account.oauth2.client.secret.<storage>.dfs.core.windows.net", "<secret>")
```

---

6. Quality Checks & Validation

Post-Write Verification:

```
# Check if data was written successfully
assert spark.read.parquet("/data/output").count() == df.count()

# Schema validation
written_schema = spark.read.parquet("/data/output").schema
assert written_schema == df.schema, "Schema mismatch!"
```

Delta Lake Data Skipping:

```
df.write.format("delta") \
  .option("delta.dataSkippingNumIndexedCols", 5) \  # Index first 5 columns
  .save("/delta/data")
```

Critical Safety Net:

```
# Fail if validation fails (e.g., empty DataFrame)
df.write.mode("overwrite") \
  .option("validate", True) \        # Validate before write (custom logic may be needed)
  .parquet("/data/output")
```

7. Real-World Use Cases

a) Time-Partitioned IoT Data:

```
sensor_data.write.partitionBy("date").parquet("s3a://iot-bucket/raw")
```

b) GDPR Compliance with Delta Lake:

```
# Delete user data and vacuum history
from delta.tables import DeltaTable
delta_table = DeltaTable.forPath(spark, "/delta/users")
delta_table.delete("user_id = 123")       # Soft delete
delta_table.vacuum(168)                   # Hard delete after 7 days
```

c) Multi-Cloud Write:

```
# Write to both S3 and GCS in parallel
df.write.parquet("s3a://bucket/data")
df.write.parquet("gs://bucket/data")
```

8. Common Pitfalls & Solutions

Issue	Solution
Small Files	Use .coalesce(n) or .repartition(n) before writing.
Accidental Overwrites	Set spark.sql.sources.partitionOverwriteMode=dynamic for partition-safe updates.
Cloud Credential Errors	Verify IAM roles/service principals and network access.
Schema Drift	Use Delta Lake schema enforcement: .option("mergeSchema", "true").

Summary

- **Output Modes**: Choose based on update strategy (append vs overwrite).
- **Formats**: Prefer Parquet/Delta for analytics; use CSV/JSON for compatibility.
- **Partitioning**: Balance file size and metadata overhead.
- **Validation**: Always verify writes in production pipelines.

Section 17: Spark SQL Integration in PySpark

1. SQL Interoperability

Concept: Bridge PySpark DataFrames and SQL for seamless querying.

a) Temporary Views:

Create session-scoped tables to query DataFrames using SQL.

```
# Create a DataFrame
data = [("Alice", 34), ("Bob", 45)]
df = spark.createDataFrame(data, ["name", "age"])

# Register as a temporary view
df.createOrReplaceTempView("people")

# Query with SQL
spark.sql("SELECT name, age FROM people WHERE age > 40").show()
```

Name	Age
Bob	45

b) Global Temporary Views:

Share views across Spark sessions but require global_temp database reference.

```
df.createGlobalTempView("global_people")

# Query from any session
spark.sql("SELECT * FROM global_temp.global_people").show()
```

Best Practice:

- Use temp views for ad-hoc analysis; avoid them in production pipelines (they're session-scoped).
- Prefer createOrReplaceGlobalTempView() for cross-session sharing in notebooks.

2. Hybrid Approach

Concept: Combine DataFrame API and SQL for readability and flexibility.

Example:

```
# Start with DataFrame
df = spark.read.parquet("/data/sales")
filtered = df.filter(col("region") == "West")

# Switch to SQL for complex joins
filtered.createOrReplaceTempView("west_sales")
enriched = spark.sql("""
  SELECT w.*, p.product_name
  FROM west_sales w
  JOIN products p ON w.product_id = p.id
""")

# Back to DataFrame for transformations
final_df = enriched.withColumn("profit", col("revenue") - col("cost"))
```

When to Use SQL:

- Complex CTEs or nested queries.
- Legacy SQL code migration.
- Team familiarity with SQL over DataFrames.

---

3. Advanced SQL Features

a) Common Table Expressions (CTEs):

Simplify complex queries with temporary result sets.

```
spark.sql("""
  WITH high_value_orders AS (
    SELECT order_id, amount
    FROM orders
    WHERE amount > 1000
  )
  SELECT c.name, h.order_id, h.amount
  FROM high_value_orders h
  JOIN customers c ON h.customer_id = c.id
""").show()
```

b) Window Functions:

Leverage SQL for ranking, running totals, and time-series analysis.

```
sql
Copy
SELECT
  product,
  month,
  SUM(sales) OVER (PARTITION BY product ORDER BY month) AS running_total FROM sales_data
```

Equivalent DataFrame Code:

```
from pyspark.sql.window import Window
window_spec = Window.partitionBy("product").orderBy("month")
df.withColumn("running_total", sum("sales").over(window_spec))
```

---

4. Catalog Integration

Concept: Programmatically manage databases, tables, and metadata.

a) Database Management:

```
# List databases
spark.catalog.listDatabases()
# [Database(name='default', description='default database', ...)]

# Set active database
spark.catalog.setCurrentDatabase("production_db")
```

b) Table Caching:

```
# Cache a table for faster access
spark.catalog.cacheTable("customers")

# Check if cached
spark.catalog.isCached("customers")  # Returns True/False
```

c) Table Inspection:

```
# List columns of a table
for column in spark.catalog.listColumns("sales"):
   print(f"Column: {column.name}, Type: {column.dataType}")

# Output:
# Column: order_id, Type: bigint
# Column: amount, Type: double
```

---

5. Performance Optimization

a) Adaptive Query Execution (AQE):

Automatically optimize shuffle partitions and joins at runtime.

```
# Enable AQE (Spark 3.0+)
spark.conf.set("spark.sql.adaptive.enabled", True)
```

b) Join Hints:

Force Spark to use specific join strategies.

```
# Broadcast join hint
spark.sql("SELECT /*+ BROADCAST(c) */ *
        FROM orders o
        INNER JOIN customers c
        ON o.cust_id = c.id")
```

c) Materialized Views (Delta Lake):
Precompute aggregates for faster queries (requires Delta Lake 1.0+):

```
spark.sql("""
  CREATE MATERIALIZED VIEW mv_product_sales
  AS SELECT product_id, SUM(amount)
  FROM sales
  GROUP BY product_id
""")
```

---

6. Security Integration
a) Row-Level Security:
Restrict access using views or Delta Lake predicates.

```
# Example: Filter rows by user role
spark.sql("""
  CREATE VIEW user_sales AS
  SELECT * FROM sales
  WHERE region = current_user()
""")
```

b) Column Masking:
Obfuscate sensitive data (supported in some distributions like Databricks):

```
spark.sql("""
  CREATE MASK ssn_mask ON TABLE employees
  FOR COLUMN ssn RETURN CONCAT('***-**-', RIGHT(ssn, 4))
""")
```

---

7. Debugging & Optimization
Query Explanation:
Analyze the physical plan to identify bottlenecks.

```
# View optimized plan
spark.sql("SELECT * FROM sales WHERE amount > 1000").explain(mode="formatted")

# Output includes:
# == Physical Plan ==
# *(1) Filter (isnotnull(amount) AND (amount > 1000.0))
# +- *(1) ColumnarToRow ...
```

Pro Tips:

- Use EXPLAIN CODEGEN to inspect generated Java code.
- Monitor SQL queries via Spark UI's **SQL** tab.

---

8. Common Pitfalls

Issue	Solution
Temp View Lifetime	Use createGlobalTempView() for cross-session access.
SQL Injection	Sanitize inputs or use parameterized queries with spark.sql() (via string formatting carefully).
Metadata Overload	Avoid excessive temporary views; use spark.catalog.dropTempView() post-use.

Summary

- **SQL + DataFrames**: Combine for readability and flexibility.
- **Catalog API**: Manage databases/tables programmatically.
- **Optimization**: Use AQE, hints, and materialized views for performance.
- **Security**: Implement row/column-level protections via views or extensions.

Section 18: Performance Tuning in PySpark

1. Critical Configuration Parameters

Objective: Optimize Spark's runtime behavior through cluster and query settings.

Key Parameters:

Parameter	Description	Best Practice
spark.sql.shuffle.partitions	Number of partitions after shuffles (e.g., joins/aggregations).	Set to **2-3x cores** in cluster (default 200). Reduce if partitions are too small (<128MB).
spark.executor.memory	Memory per executor.	Allocate 70% of node memory (e.g., 8G for 12G node). Leave 1-2G for OS/buffer.
spark.dynamicAllocation.enabled	Allow Spark to scale executors dynamically.	Enable (true) for variable workloads. Set spark.dynamicAllocation.minExecutors and maxExecutors.
spark.sql.adaptive.enabled	Let Spark optimize shuffle partitions at runtime (AQE).	Always enable (true) for Spark 3.0+.

Example Configuration:

```
configs = {
   "spark.sql.shuffle.partitions": "200",      # Default, adjust based on data size
   "spark.executor.memory": "8g",           # For 12G node
   "spark.driver.memory": "4g",             # Driver-side operations (e.g., collect())
   "spark.dynamicAllocation.enabled": "true",   # Scale executors as needed
   "spark.sql.adaptive.coalescePartitions.enabled": "true"  # Merge small shuffle partitions
}
spark = SparkSession.builder.config(conf=configs).getOrCreate()
```

2. Partition Optimization

a) Ideal Partition Size:

- **Goal**: Balance parallelism and overhead.
- **Rule of Thumb**: 128MB–1GB per partition.
- **Calculate Partitions**:

```
df_size_gb = df.rdd.map(lambda x: len(str(x))).sum() / (1024**3)  # Approx. size
desired_partitions = max(1, int(df_size_gb / 0.5))  # 500MB partitions
df = df.repartition(desired_partitions)
```

b) Handling Skewed Data:

Problem: Uneven data distribution (e.g., 90% of data in one key).

Solution: Salting – Add a random prefix to keys to redistribute.

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

# Add salt (0-9) to skewed key
df = df.withColumn("salted_key", col("original_key").cast("str") + "_" + (rand() *
10).cast("int").cast("str"))

# Repartition and aggregate
df.repartition("salted_key") \
  .groupBy("original_key") \
  .agg(sum("value").alias("total_value"))
```

3. Caching Strategies

When to Cache:

- Datasets reused in multiple actions (e.g., loops, iterative ML).
- Small lookup tables for frequent joins.

Storage Levels:

Level	Description	Use Case
MEMORY_ONLY	Store in memory (no serialization).	Fast access for small datasets.
MEMORY_AND_DISK	Spill to disk if memory full.	Large datasets (>60% of memory).
DISK_ONLY	Store only on disk.	Rarely used; avoid unless necessary.

Example:

```
from pyspark import StorageLevel

# Cache with spill-to-disk
df.persist(StorageLevel.MEMORY_AND_DISK)

# Verify caching
if spark.catalog.isCached("table_name"):
   print("Cached!")

# Release memory
df.unpersist()
```

4. Join Optimization Techniques

a) Broadcast Join:

- **Use Case**: Small table (≤10MB) + large table.
- **Implementation**:

```
from pyspark.sql.functions import broadcast
df_large.join(broadcast(df_small), "key")
```

b) Sort-Merge Join:

- **Use Case**: Large tables sorted on join keys.
- **Implementation**:

```
df1.hint("merge").join(df2, "key")
```

c) Bucket Join:

- **Use Case**: Frequent joins on the same column.
- **Steps**:
    1. Pre-bucket tables:

```
df.write.bucketBy(50, "key").sortBy("key").saveAsTable("bucketed_table")
```

    2. Join without shuffle:

```
spark.sql("SELECT * FROM bucketed_a a JOIN bucketed_b b ON a.key = b.key")
```

---

5. Debugging Tools

a) Execution Plans:

- **Logical Plan**: High-level transformations.
- **Physical Plan**: Actual execution steps (e.g., scan, filter, exchange).

```
df.explain(mode="extended")  # Show all plans (parsed, analyzed, optimized, physical)
```

b) Spark UI:

- Access at http://<driver-ip>:4040.
- **Key Features**:
    - **Jobs Tab**: Identify slow stages.
    - **Stages Tab**: View shuffle read/write sizes.
    - **SQL Tab**: Analyze query execution DAG.

---

6. File-Level Optimizations

a) File Size Control:

```
df.write.option("maxRecordsPerFile", 100000)  # ~1GB at 1KB/record
  .parquet("path")
```

b) Parquet Tuning:

```
df.write.option("parquet.block.size", 256 * 1024 * 1024)  # 256MB row groups
  .parquet("path")
```

c) Statistics Collection:

spark.sql("ANALYZE TABLE sales COMPUTE STATISTICS FOR COLUMNS product_id, revenue")

- Enables better query planning (e.g., join order).

---

7. Real-World Use Cases

a) Skewed Aggregation:

- **Problem**: 80% of sales come from 5% of customers.
- **Solution**: Salting + two-stage aggregation.

b) Iterative ML Pipeline:

- Cache feature-engineered DataFrames reused across training cycles.

c) Large Joins:

- Use bucket joins for daily fact-dimension table merges.

---

8. Common Pitfalls

- **Too Many Partitions**: Increases task scheduling overhead.
- **Over-Caching**: Wastes memory on rarely used data.
- **Ignoring AQE**: Failing to enable Adaptive Query Execution in Spark 3+.

Section 19: Delta Lake Integration in PySpark

1. Key Features of Delta Lake

Concept: Delta Lake is an open-source storage layer that adds reliability, ACID transactions, and data management to data lakes.

Feature	Description	Use Case
ACID Transactions	Ensures atomic commits and isolation between concurrent reads/writes.	Preventing partial writes during ETL failures.
Time Travel	Access historical versions of data using version numbers or timestamps.	Auditing, rollbacks, and reproducing past reports.
Schema Enforcement	Rejects writes that don't match the table schema.	Data quality control (e.g., blocking nulls in a required column).
Upserts	Update, delete, or merge data using SQL-like MERGE INTO syntax.	Change Data Capture (CDC) pipelines.

2. Critical Operations

a) Creating a Delta Table:

```
from pyspark.sql import SparkSession

spark = SparkSession.builder \
   .appName("DeltaDemo") \
   .config("spark.sql.extensions", "io.delta.sql.DeltaSparkSessionExtension") \
   .config("spark.sql.catalog.spark_catalog", "org.apache.spark.sql.delta.catalog.DeltaCatalog") \
   .getOrCreate()

# Write DataFrame as Delta table
df.write.format("delta") \
  .mode("overwrite") \
  .option("delta.logRetentionDuration", "30 days")  # Keep transaction log for 30 days \
  .save("/data/delta/sales")
```

b) Schema Evolution:

```
python
Copy
# Allow new columns to be added automatically
spark.conf.set("spark.databricks.delta.schema.autoMerge.enabled", True)

# Append data with new columns
new_df.write.format("delta") \
  .mode("append") \
  .option("mergeSchema", "true") \  # Merge schemas during write
  .save("/data/delta/sales")
```

c) Time Travel:

```
# Read data from version 5 (using version number)
df_v5 = spark.read.format("delta") \
  .option("versionAsOf", 5) \
  .load("/data/delta/sales")

# Read data as of a timestamp
df_timestamp = spark.read.format("delta") \
  .option("timestampAsOf", "2024-01-01") \
  .load("/data/delta/sales")

# Compare two versions (e.g., audit changes)
spark.sql("""
  SELECT * FROM delta.`/data/delta/sales` VERSION AS OF 12
  EXCEPT
  SELECT * FROM delta.`/data/delta/sales` VERSION AS OF 15
""")
```

---

3. Performance Optimizations

a) Z-Ordering:

Optimize data layout to speed up queries on specific columns.

```
spark.sql("OPTIMIZE delta.`/data/delta/sales` ZORDER BY (customer_id, date)")
```

- **Benefit**: Co-locates related data (e.g., all records for a customer) in the same files.

b) Compaction (Bin-Packing):

Merge small files to improve read efficiency.

```
spark.sql("OPTIMIZE delta.`/data/delta/sales`")
```

c) Vacuum Old Files:

Remove unused files older than retention period.

```
spark.sql("VACUUM delta.`/data/delta/sales` RETAIN 168 HOURS")  # Keep 7 days of history
```

---

4. DML Operations

a) Upsert with MERGE INTO:

```
# Target table: delta_sales | Source table: updates

spark.sql("""
  MERGE INTO delta.`/data/delta/sales` AS target
  USING updates AS source
  ON target.order_id = source.order_id
  WHEN MATCHED THEN
    UPDATE SET target.amount = source.amount, target.status = source.status
  WHEN NOT MATCHED THEN
    INSERT (order_id, amount, status) VALUES (source.order_id, source.amount, source.status)
""")
```

b) Change Data Feed (CDF):

Track row-level changes for CDC pipelines.

```
# Enable CDF during table creation
df.write.format("delta") \
  .option("delta.enableChangeDataFeed", "true") \
  .save("/data/delta/sales")

# Read changes between versions 5 and 10
changes_df = spark.read.format("delta") \
  .option("readChangeFeed", "true") \
  .option("startingVersion", 5) \
  .option("endingVersion", 10) \
  .load("/data/delta/sales")
```

5. Audit and Compliance

a) View History:

```
history_df = spark.sql("DESCRIBE HISTORY delta.`/data/delta/sales`")
history_df.select("version", "timestamp", "operation", "operationParameters").show()

# Output:
# +-------+-------------------+---------+----------------------+
# |version|        timestamp|operation|  operationParameters|
# +-------+-------------------+---------+----------------------+
# |      2|2024-03-20 12:30:45|   MERGE |{predicate -> "..."} |
# +-------+-------------------+---------+----------------------+
```

b) Rollback:

```
# Restore to version 8
spark.sql("RESTORE delta.`/data/delta/sales` TO VERSION AS OF 8")
```

c) Retention Policies:

```
# Set defaults for new tables
spark.conf.set("spark.databricks.delta.properties.defaults.logRetentionDuration", "30 days")
spark.conf.set("spark.databricks.delta.properties.defaults.deletedFileRetentionDuration", "15 days")
```

---

6. Real-World Use Cases

a) GDPR Compliance:

- Use MERGE INTO to delete/update user records.
- Leverage time travel to audit deletions.

b) Machine Learning Reproducibility:

- Train models on specific data versions.
- Roll back to previous dataset states if needed.

c) Streaming Pipelines:

- Combine with Spark Structured Streaming for exactly-once processing.
- Use CDF to trigger downstream workflows.

---

7. Delta Lake vs. Parquet

Feature	Delta Lake	Parquet
ACID Transactions	✅	❌
Schema Evolution	✅ (Auto-merge)	❌ (Manual Hive)
Time Travel	✅	❌
Upserts	✅ (MERGE)	❌ (Overwrite only)
Performance	Optimized (Z-order)	Basic

---

8. Common Pitfalls

Issue	Solution
Unmanaged Retention	Always set logRetentionDuration to avoid unbounded log growth.
Schema Drift	Use mergeSchema cautiously; prefer explicit schema evolution.
Small Files	Schedule daily OPTIMIZE jobs.

Summary

- **Delta Lake** adds enterprise-grade reliability to data lakes.
- Key operations: Time travel, schema evolution, and MERGE for CDC.
- Optimize with Z-ordering, compaction, and CDF for streaming.

Section 20: Real-World Project Walkthrough

Retail Analytics Pipeline Architecture
Objective: Process 100GB/day of sales data with validation, incremental updates, and fault tolerance.
Architecture: Medallion (Bronze → Silver → Gold)
Tools: PySpark, Delta Lake, AWS S3, Power BI

---

1. Bronze Layer (Raw Data Ingestion)
Goal: Ingest raw CSV files with schema validation and auditability.

```
# Define schema to enforce data quality
bronze_schema = """
  invoice_id STRING,
  store_id INT,
  timestamp TIMESTAMP,
  items ARRAY<STRUCT<
    product_id: STRING,
    quantity: INT,
    price: DECIMAL(10,2)
  >>,
  _ingest_time TIMESTAMP  # Audit column
"""

(spark.readStream
  .schema(bronze_schema)  # Block invalid data upfront
  .option("header", "true")
  .csv("s3://raw-bucket/")
  .withColumn("_ingest_time", current_timestamp())  # Track ingestion time
  .writeStream
  .format("delta")
  .option("checkpointLocation", "/checkpoints/bronze")  # Exactly-once guarantee
  .outputMode("append")
  .start("s3://bronze-layer/sales"))
```

Key Features:

- **Schema Enforcement**: Rejects malformed records during ingestion.
- **Checkpointing**: Tracks progress and ensures fault tolerance.
- **Audit Column**: _ingest_time helps debug data freshness issues.

---

2. Silver Layer (Cleaned Data)
Goal: Clean, validate, and explode nested data for analysis.

```
def process_silver(batch_df, batch_id):
  # Data Quality Checks
  batch_df = batch_df.filter(
    (col("store_id").isNotNull()) &  # Reject null stores
    (size(col("items")) > 0)       # Reject empty orders
  )

  # Explode items array into individual rows
  exploded = batch_df.select(
    "invoice_id",
    "store_id",
    explode("items").alias("item")  # Split array into rows
  )

  # Write to Delta with schema evolution
  exploded.write.mode("append") \
    .format("delta") \
    .option("mergeSchema", "true")  # Auto-add new columns
    .save("s3://silver-layer/sales")

# Stream from Bronze to Silver
spark.readStream.format("delta") \
  .load("s3://bronze-layer/sales") \
  .writeStream \
  .foreachBatch(process_silver) \  # Custom batch processing
  .option("checkpointLocation", "/checkpoints/silver") \
  .start()
```

Optimizations:

- **Filter Early**: Remove invalid data before transformations.
- **Explode Nested Data**: Prepare for aggregation in the Gold layer.
- **Schema Evolution**: Automatically handle new fields (e.g., promo codes).

3. Gold Layer (Aggregated Metrics)

Goal: Compute daily store metrics with incremental updates.

```
# MERGE INTO for efficient upserts (existing dates get updated)
spark.sql("""
 MERGE INTO gold.daily_store_metrics target
 USING (
  SELECT
   store_id,
   date(timestamp) as date,
   SUM(item.price * item.quantity) as revenue,
   COUNT(DISTINCT invoice_id) as transactions
  FROM silver.sales
  WHERE timestamp > current_date() - INTERVAL 1 DAY  # Incremental processing
  GROUP BY store_id, date
 ) source
 ON target.store_id = source.store_id AND target.date = source.date
 WHEN MATCHED THEN UPDATE SET *
 WHEN NOT MATCHED THEN INSERT *
""")

# Optimize layout for dashboard queries
spark.sql("OPTIMIZE gold.daily_store_metrics ZORDER BY (date)")
```

Key Features:

- **Incremental Processing**: Only process the latest day's data.
- **Z-Ordering**: Co-locates data by date for faster time-range queries.
- **ACID Compliance**: MERGE INTO ensures updates are atomic.

---

4. Production Best Practices

a) Monitoring & Alerting:

```
# Track active streams
for stream in spark.streams.active:
  print(f"Stream ID: {stream.id}, Status: {stream.status}")

# Alert on failure using try/catch
try:
  query.awaitTermination()
except StreamingQueryException as e:
  send_slack_alert(f"Pipeline failed: {e}")
  # Auto-restart using checkpoint
```

b) CI/CD Integration (Databricks Example):

```
job_spec = {
   "name": "nightly-sales-pipeline",
   "tasks": [{
      "task_key": "silver-layer",
      "notebook_path": "/Production/Silver_Processing",
      "timeout_seconds": 3600,
      "retry_on_timeout": True  # Auto-retry transient errors
   }]
}
```

c) Post-Deployment Checks:

1. **Validate Outputs**:

```
spark.sql("ANALYZE TABLE gold.daily_store_metrics COMPUTE STATISTICS")
```

2. **Retention Policies**:

```
spark.sql("VACUUM gold.daily_store_metrics RETAIN 90 DAYS")  # Comply with GDPR
```

3. **Cost Control**:

```
spark.sql("SET spark.databricks.delta.optimize.maxFileSize = 134217728")  # 128MB files
```

---

5. Debugging Toolkit

a) Data Lineage:

```
# Track table evolution
display(spark.sql("DESCRIBE HISTORY gold.daily_store_metrics"))
```

b) Query History:

```
# Audit job performance
display(spark.sql("SELECT * FROM system.operational_metrics.jobs"))
```

c) Data Sampling:

```
# Debug without full scans
spark.read.option("samplingRatio", 0.01).format("delta").load("s3://gold-layer")
```