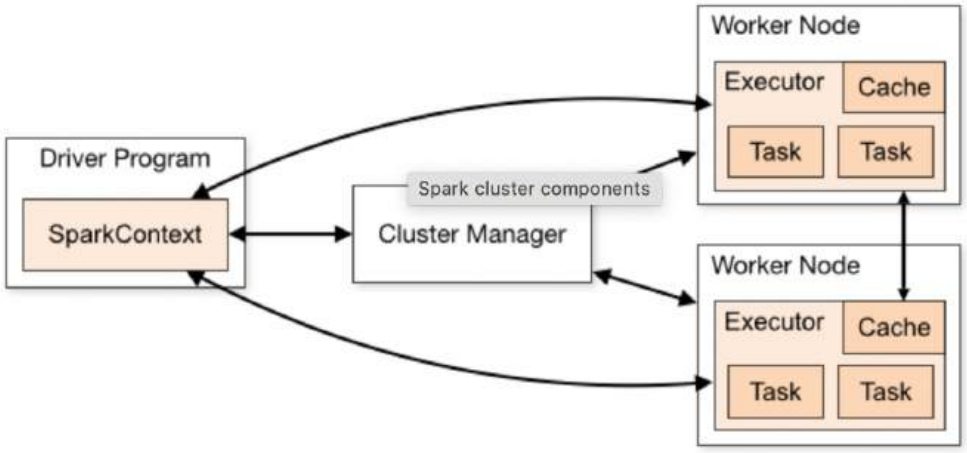
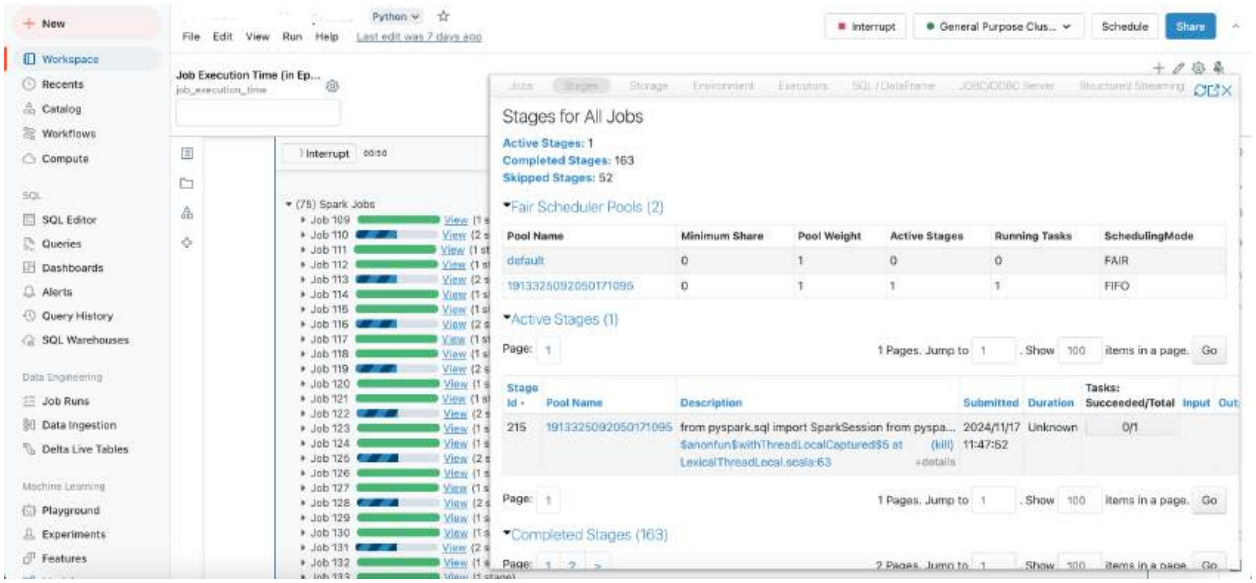
PySpark

Batch Pipelines





- Spark turns the user's data processing commands into a Directed Acyclic Graph, or DAG, which determines the execution plan, i.e what tasks are executed on what nodes and in what sequence.
- Spark runs in a distributed fashion by combining a Driver core process that splits a Spark application into tasks & distributes them among many Executor processes that do the work. Driver sends tasks to executors via Cluster Manager.
- OpenSource vs Databricks managed Spark:

    - Native Spark (open-source): Client spark submits job → Driver JVM starts → Resource Manager (YARN/Mesos/K8s) allocates resources (cores, memory) to driver & executors. Jobs, Stages, Tasks executed. You must configure: driver-memory, driver-cores, num-executors, executor-memory, executor-cores, dynamic allocation, cluster manager settings. You can run Resource Manager on same master node.
    - Databricks (managed Spark): User submits job → Databricks handles almost EVERYTHING. You only configure high-level things.
- Data Processing:
    - Resilient Distributed Dataset (RDD): Low-level abstraction, compile-time checking before the job runs
      (i.e, type-safety,
      eg: val df = Seq((1,"Alice")).toDF("id", "name")
      df.select("nam").show()) // Typo in column name: "nam" instead of "name". // Compilation error thrown in start, before job runs
      ), NOT Catalyst/Tungsten engine optimised hence comparatively slow, Use - Lowlevel data transformations / custom partitioning/data not fitting tabular structure.
    - DataFrame: Medium-level abstraction, NO compile-time checking, Catalyst/Tungsten optimised, Use - Analytical queries, SQL-like aggregations, joins, filters.
    - Dataset: High-level abstraction, Compile-time checking, Catalyst/Tungsten optimised, Use - Type safety + functional + SQL benefits (Scala/Java only).
- When Spark runs distributed code, it must serialize data and code before sending to worker nodes.
    - RDD stores raw JVM/Python objects, each record is a normal object ((Int, String), class, etc.). To shuffle or persist these, Spark must: Serialize each object to bytes -> Send across the cluster -> Deserialize on the other side. This adds overhead.
    - DataFrames and Datasets are engine optimized, i.e storing data in binary tungsten format than usual java/pickle serialized format. Again, serialization/deserialization not needed.

- Spark does lazy evaluation, i.e transformations don't execute immediately. They build/add nodes in the logical plan (DAG), and only actions trigger actual computation and optimizations by engines during runtime. Jobs->Stages->Tasks.
- Components: Driver node creates jobs/stages/tasks, and workers execute or process data partitions.
    - Application - A user program built on Spark using its APIs. It consists of a driver program and executors/workers on the cluster.
    - Job - Refers to a sequence of transformations on data in response to a Spark Action. Each job is converted into a DAG with execution plan/stages.
        - Execution: Sequential, unless you explicitly trigger multiple actions asynchronously (e.g., via threads, async APIs, or multiple Spark contexts).
        - Actions: collect(), count(), take(n)/head(n)/tail(), first(), foreach(func), foreachPartition(func), saveAsTextFile(), saveAsParquet(), saveAsTable(), reduce(func), fold(), aggregate(), countByKey(), collectAsMap(), toPandas(), write.format(...).save(), show(), rdd.takeSample(), rdd.saveAs...(), spark.sql("SELECT ...").show(), writeStream.start().
            - spark.read.format()- just builds a logical plan, no execution. But if you combine it with spark.read.parquet("path").show(), then:
                - Job 1: Schema inference. It may scan some part of data/ read metadata to infer schema.
                - Job 2: Actual action (show, count, etc.)
                - Exception: If schema is already known (Delta, Hive table), only 1 job happens.
            - Similarly spark.sql("SELECT ...") will only build a logical plan, no action. But if you couple it with .show(), then action is triggered.
        - Case:
            - count(), it triggers a job, computation happens on workers, and later aggregation of stats happens on the

driver - it's a safe function generally for heavy data also. But directly using collect(), it will bring rows on the driver which can cause an OOM on driver.
        - `head()` - Action (efficient - early termination possible). `tail()` - Action (less efficient - must scan entire dataset)
- Stage - Each job gets divided into smaller sets of tasks called stages, in which a sequence of transformations can be executed in a single pass, i.e., without any shuffling of data. The boundary between two stages is due to data shuffling. Transformations never “bring data” immediately (they’re lazy). But they define how tasks and stages will run once an action triggers them.
    - Execution: Sequential
    - Transformations:
        - Wide (create a new stage): reduceByKey(), groupByKey(), join(df, on="id", how="left"), distinct(), repartition(), sortBy(), orderBy(), sortWithinPartitions(), coalesce(numPartitions, shuffle=True), cogroup(), cartesian(), aggregateByKey().
        - Narrow: map(), filter(), union(), flatMap(), withColumn(), drop() / alias(), mapPartitions(), mapValues(), sample(), keyBy(), select() / where() / limit(), cache() / persist() / checkpoint(), createOrReplaceTempView().
            - limit(n) is tricky: narrow if no global order, wide if combined with orderBy.
- Task - A single unit of work or execution that will be sent to a Spark worker/executor. Stage divided into tasks, i.e: when a stage comprises transformations on an RDD, those transformations are packaged into a task to be executed on a single executor. Each partition in your RDD or DataFrame is processed by exactly one task, which is mapped to a single core. You may have many tasks, but how many of them execute in parallel depends on the underlying infra, i.e cores.
    - Execution: Parallel
- Example:

```
spark = SparkSession.builder \
    .appName("MultipleJobsStagesTasksExample") \
    .master("local[4]") \  # 4 cores, so up to 4 tasks can run in
parallel
    .config("spark.executor.memory", "2g") \
    .config("spark.driver.memory", "1g") \
    .config("spark.sql.shuffle.partitions", "4") \  # each shuffle
will create 4 tasks
    .getOrCreate()

df = spark.createDataFrame(data, columns)
## Repartition to 8 partitions (creates tasks)
df = df.repartition(8, "group_id")
## TRANSFORMATIONS (lazy)
df_filtered = df.filter(col("amount") > 20) # narrow
transformation, no new stage
df_mapped = df_filtered.withColumn("amount_plus_5", col("amount")
+ 5)
## WIDE TRANSFORMATION (shuffle triggers new stage)
df_grouped =
df_mapped.groupBy("group_id").agg(avg("amount_plus_5").alias("avg_
amount"))
## ACTION 1: triggers Job 1
df_grouped.show()
## Another ACTION: triggers Job 2
df_mapped.count()
—-----------------------------------------------
Job 1 → df_grouped.show()
  Stage 1: read & filter & map (df_filtered, df_mapped) → narrow
transformations → tasks = 8 (Usual 1 task per partition, so 8
tasks since 8 partitions)
  Stage 2: groupBy (shuffle) → tasks = 4
Driver collects final result to print (show())

Job 2 → df_mapped.count(): New action triggers another job
  Only narrow transformations → one stage (no shuffle) -> 8 tasks
(since one task per partition) count rows
```

```
  Partial counts sent to driver → aggregated → final count

** Had it been df_grouped.count() rather than df_mapped, then same
number of stages/tasks would be created.
```

- Note:
    - Cache intermediate DataFrames to avoid recomputation across stages. In above example: caching/persist would help save time, BUT note that the number of stages/tasks created would be in expected lines and not reduced.
        - Cache vs Persist vs Checkpoint:
            - Cache: Stores the DataFrame in memory (RAM) by default. It's equivalent to: df.persist(StorageLevel.MEMORY_ONLY). To remove cache ~ df.unpersist()
            - Persist: More flexible version of cache() — you can choose where and how to store data: Memory only, Memory + Disk, Disk only, Serialized, etc.
            - Checkpoint: It's like a savepoint. Breaks the lineage of a DataFrame (cuts off its dependency graph); Writes data to disk (HDFS/S3) as a new materialized copy; Used for fault tolerance and state management (streaming or long transformations). Forces a job execution. Eg: spark.sparkContext.setCheckpointDir("/mnt/checkpoints"), then df.checkpoint().
    - config("spark.sql.shuffle.partitions", "4") is not the same as .repartition(4). The config sets how many partitions Spark creates after any shuffle (a default). Repartition forces a shuffle immediately and creates exactly 4 partitions for that DataFrame.
    - Job/stage planning happens on driver node. Task execution on worker nodes.
    - Actions are eager evaluation, transformations are lazy.
- A partition is a logical chunk of data that Spark processes in task on a core. Ideal: 2-3 partitions on a core.

- Default partitioning: Spark automatically partitions data based on the underlying file system block size — typically 128 MB or 256 MB per block in HDFS/S3. Eg:
    - Reading 3 files of 1 GB → ≈ 8 × 3 = 24 partitions (1 GB / 128 MB ≈ 8).
    - Reading 3 small files of 10 MB → only 3 partitions (min 1 per file).
    - By default, Spark uses Hash Partitioning or Range Partitioning depending on operation type (e.g., joins use hash, sorts use range).
    - Spark recommends 2–3 tasks per CPU core for optimal parallelism.
- Adjust partitions using:
    - repartition(n) – increase/decrease partitions with full shuffle (expensive but creates balanced, equal-sized partitions).
    - coalesce(n) – reduce partitions without full shuffle (cheap but may create uneven partitions; good for writing smaller outputs).
        - Use coalesce(..., shuffle=True) to rebalance when reducing partitions safely. Default False.
    - PartitionBy (When Writing Data) - write.partitionBy("country") → creates subfolders like country=IN/, country=US/ in output path. Great for filter pushdown and query pruning in Hive/Glue tables (will discuss).
- Partition strategy:
    - Count: If high -> excessive overhead in managing many small tasks, If low -> underutilised cores in cluster.
    - Size: If large -> long computation time, slow write times, executor/worker OOM, If small -> slow read time downstream, large task creation overhead, driver OOM
- Avoid partition on high-cardinality keys (e.g., user_id) → causes small files problem, skew, catalogue explosion, and slow query planning.
- Types of Partitioning:

        - Hash Partitioning: Takes your key, applies a hash function, and assigns it to a partition by: partition = hash(key) % numPartitions. Deterministic distribution but if one key repeats a lot, that partition maybe skewed. Eg: join(), groupByKey(), reduceByKey(), distinct(). Code ex:
          rdd = sc.parallelize([("A", 1), ("B", 2), ("A", 3), ("C", 4)], numSlices=3)
          grouped = rdd.groupByKey() # triggers hash partitioning
        - Range Partitioning: Sorts keys and divides them into contiguous ranges, so each partition gets a range of keys, say: partition 0: keys < 100, partition 1: 100–199, etc. Eg: sortByKey(), Range-based joins, etc.
          Code ex:
          data = [(5, "A"), (1, "B"), (10, "C"), (20, "D"), (100, "E")]
          rdd = sc.parallelize(data, 2)
          sorted_rdd = rdd.sortByKey(numPartitions=3) # internally uses RangePartitioner
        - Custom Partitioning: By extending the Partitioner class (RDD-level only, not DataFrame).
- Shuffling is the process of exchanging data between partitions. It's an expensive operation. Triggered by wide-transformations.
    - Order of cost: ORDER BY (most expensive) > JOIN > GROUP BY.
    - Shuffle Optimization:
        - Broadcast joins – small DataFrame broadcasted to all executors which avoids shuffle for the large dataset. Eg:

```
joined = large_df.join(broadcast(small_df), "id", "left")
```

        - Salting – add random prefixes to skewed keys before join/group. Eg:

```
df1 = spark.createDataFrame([("A", 10), ("A", 20), ("A", 30),
("B", 5)], ["key", "val"])
df2 = spark.createDataFrame([("A", "infoA"), ("B", "infoB")],
["key", "info"])
# Salt the skewed key 'A'
salted_df1 = df1.withColumn("salt", F.when(F.col("key") == "A",
```

```
(F.rand() * 3).cast("int")).otherwise(0))
salted_df1 = salted_df1.withColumn("salted_key",
F.concat(F.col("key"), F.lit("_"), F.col("salt")))

salts = spark.range(3).toDF("salt")
salted_df2 = df2.join(salts, how="cross").withColumn("salted_key",
F.concat(F.col("key"), F.lit("_"), F.col("salt")))

joined = salted_df1.join(salted_df2, "salted_key")
```

- Bucketing – pre-partition large datasets by join key, avoids reshuffling.

```
spark.conf.set("spark.sql.sources.partitionOverwriteMode",
"dynamic")

df1 = spark.createDataFrame([(1, "A"), (2, "B"), (3, "C")], ["id",
"x"])
df2 = spark.createDataFrame([(1, "a"), (2, "b"), (3, "c")], ["id",
"y"])

# Write both as bucketed tables (e.g., 4 buckets on id)
df1.write.bucketBy(4,
"id").sortBy("id").saveAsTable("bucketed_df1")
df2.write.bucketBy(4,
"id").sortBy("id").saveAsTable("bucketed_df2")

bucketed1 = spark.table("bucketed_df1")
bucketed2 = spark.table("bucketed_df2")

joined = bucketed1.join(bucketed2, "id")
# Data for same id always lands in same bucket. Spark reads
aligned buckets → no shuffle. Very efficient for repetitive joins
on same key.
```

- Repartition by key – ensures data co-location.

```
df1p = df1.repartition("id")
df2p = df2.repartition("id")
```

```
joined = df1p.join(df2p, "id")
# Each partition now holds a consistent key range/hash. Shuffle
happens once, not redundantly.
```

- Filter early – apply filters before joins/aggregations.
- Use map-side combine (reduceByKey vs groupByKey).

```
When you do an aggregation (like reduceByKey, countByKey,
sumByKey, etc.), Spark tries to reduce the amount of data shuffled
across the network. It does this by aggregating partial results
locally on each executor before the shuffle -- that's called
map-side combine. This optimization happens automatically with
reduceByKey, but not with groupByKey.
Eg:
Case 1: result = rdd.groupByKey().mapValues(sum) -> Each executor
collects all values per key → sends all (key, value) pairs over
network to one reducer. Reducer receives all those values and sums
them. All values are shuffled -- if "A" appears a million times,
all 1 million pairs for "A" go over the network.
Case 2: result = rdd.reduceByKey(lambda a, b: a + b) -> Each
executor locally computes partial sums before shuffle. Reducer
receives small number of partial results like [('A', 6), ('B', 9)]
and merges them. Much less data shuffled -- Spark has already
"pre-aggregated" locally.
```

- Data Skew Optimization: Salting, Repartition, Enable Skew join hints with SKEW(df), etc.
- Framework-level/Other optimizations:
    - Catalyst Optimizer (Logical Optimization Engine):
        - Catalyst = rule-based tree transformation framework. Converts user query → optimized physical plan. SQL query plan is a tree of nodes, which is optimized based on rules.
        - Unresolved Logical Plan → Analyzed Logical Plan → Optimized Logical Plan → Physical Plan (cost-based)
            - Eg: SELECT name FROM users WHERE age > 5 AND 1 = 1

            - Optimized Logical Plan: Project [name]-> Filter [age > 30] -> Relation users; Catalyst removes 1=1, pushes filter below projection.
        - Stages:
            - Analysis – resolve references, check schema, infer data types.
            - Logical Optimization – apply rules:
                - Predicate pushdown (filter early): Reads only relevant rows from storage, like: WHERE age > 30
                - Projection pushdown (select few columns): Reads only required columns, like: SELECT name, age
                - Constant folding: replaces constant expressions at compile time (1 + 2 → 3).
                - Null propagation: replaces expressions with null if inputs are null.
                - Boolean simplification
                - Reordering joins: reorders join sequence based on estimated size.
            - Physical Planning – choose best physical plan (e.g., broadcast join vs shuffle join).
                - Configurations like: spark.sql.autoBroadcastJoinThreshold, spark.sql.shuffle.partitions, spark.sql.adaptive.enabled influence the physical plan. You can plug in: custom logical rules, optimizer rules, parser injection.
            - Code Generation – hand off optimized plan to Tungsten engine.
    - Tungsten Engine (Physical Execution Layer):
        - Executes the optimized plan from Catalyst. Focuses on memory & CPU efficiency.
        - Stages:
            - Whole-Stage Code Generation: Instead of interpreting every row through function calls (Row → apply → next

→ Row), Spark generates compiled Java code for an entire physical query stage. At high level:

```
Spark uses a codegen template in Scala/Java for each operator
(FilterExec, ProjectExec, etc.).
The CodegenSupport interface lets these operators provide snippets
of Java source code.
At runtime, Spark merges all snippets of operators in a stage →
generates one long Java class (as a string) → compiles it into
bytecode using Janino (a lightweight JIT compiler).
This compiled bytecode runs directly on JVM without function call
overheads. Eg:
Old:
while (iterator.hasNext()) {
   val row = iterator.next()
   if (row.value > 10) {
       val newRow = row.value * 2
       output.append(newRow)
   }
}
New:
for (int i = 0; i < numRows; i++) {
   if (column[i] > 10) output[i] = column[i] * 2;
}
```

- Off-Heap Memory Management: Spark bypasses the JVM heap and manages memory natively, similar to C/C++. Uses sun.misc. Unsafe or Platform APIs to allocate memory manually. Instead of Java ArrayList<Row>, Spark has a flat binary block in native memory → each field accessed by offset.
- Cache-Aware Execution: Stores serialized data in contiguous memory regions. Uses UnsafeRow format → all columns of a row stored in one contiguous memory block.
- Vectorized Processing: Operates on batches of rows (columnar) instead of row-by-row (iterator model).

Introduced with Parquet/ORC readers (VectorizedParquetReader).
        - Parquet is columnar format data storage.
- Adaptive Query Execution (AQE):
    - Dynamically optimizes physical plans at runtime based on real data stats.
    - Stages:
        - Re-optimize join strategies (switch shuffle ↔ broadcast, Spark initially plans a Shuffle Join. During execution, detects if has to switch).
        - Merge small partitions → fewer tasks.
        - Split skewed partitions. (Detects one shuffle partition much larger than others. Splits that partition into smaller ones to avoid single strangler task.)
        - Optimize shuffle partition count automatically (If Spark planned 200 partitions but actual data small, AQE merges to fewer partitions (say 10) → reduces overhead.)
- Cost-Based Optimizer (CBO):
    - Uses collected table statistics (size, NDV, histograms) to choose better join orders and physical plans, works with Catalyst. Enabled via: spark.conf.set("spark.sql.cbo.enabled", True)
- Executor Tuning:
    - Each Spark executor is a JVM process that runs multiple tasks in parallel, sharing CPU cores and memory. Tuning it effects the data processing.
        - Fat Executors: Few executors with more cores & memory per executor
            - Pros: Low shuffle, context switching, startup overhead.
            - Cons: More GC time, less parallelism.
            - Use in small cluster.
        - Tiny Executors: Many executors with fewer cores & memory each

        - Pros: High parallelism, localised failures.
        - Cons: High shuffle, Scheduler overhead.
        - Use in large cluster.
    - Use balanced executors.
- ZOrder Optimization:
    - Z-Ordering is a multi-dimensional clustering technique for data files (typically Parquet/Delta). It co-locates related column values together in the same set of files to make filtering and range scans faster.

```
Eg:
SELECT * FROM transactions WHERE country = 'IN' AND event_time
BETWEEN '2025-10-01' AND '2025-10-10';

Databricks physically rearranges data on disk so that rows with
similar country and event_time values end up close together (in
the same files).
So when you filter by these columns, Databricks needs to read
fewer files — because the relevant data is tightly clustered
instead of being scattered across many files.
Query: OPTIMIZE transactions ZORDER BY (country, event_time);

Internally, it interleaves the bits and stores, like:
country bits:   0 1 1 0 0 1
event_time bits:1 1 1 0 1 0 0 1 0 1 0 1
Z-order bits:   0 1 1 1 0 0 1 1 0 0 1 0 1 0 1…
Rows are sorted by this interleaved single composite Z-order bit
key.
Z-Ordering > Sorting by One Column
```

- Code/SQL Query tips:
    - UNION removes duplicate rows, UNION ALL does not. There is a performance hit when using UNION instead of UNION ALL, choose based on usecase.
    - Avoid collect()/toPandas(). Prevents OOM & driver crashes.
    - Use coalesce() to reduce partitions.
    - Use native Spark SQL functions over UDFs.

- Spark UDFs: UDF (User-Defined Function) is a custom function written by the user to extend Spark's capabilities beyond the built-in functions provided by Spark SQL. It runs row by row on executors hence slow.
- Scala/Java/SQL UDF - Catalyst optimized, Pandas UDF - partially optimized, Python UDF - Executes via Py4J, not optimised as serialization overhead.
- Note that print() or debug() logs may not be visible as UDFs run on workers, not drivers to view logs. And when UDF fails on one partition, the entire stage fails and is retried.

```
@udf(returnType=IntegerType())
def safe_divide(a, b):
    try:
        return a // b if b != 0 else None
    except Exception:
        return None
```

- Filtering before joins, AQE, skews fix, etc.
- spark.sql.shuffle.partitions default value is 200, plan accordingly.
- Use G1GC or Parallel GC for large heaps (Details in Java page).
- Use KryoSerializer instead of Java serializer.
    - When Spark runs distributed jobs: Data is sent between executors (e.g., for shuffles, caching, broadcasts). Each time, Spark must serialize (convert objects into byte form) to transfer them across the network or store them in memory/disk. Hence performance matters.
- Prefer partition pruning > filter at runtime.
- Use checkpoint when lineage too long.
- EXISTS/NOT EXISTS (Stops evaluating as soon as a match is found, faster) > IN/NOT IN (Subquery may be fully evaluated, slower on large datasets).

```
SELECT * FROM orders
```

```
WHERE customer_id IN (SELECT id FROM customers WHERE active = 1);

SELECT * FROM orders o
WHERE EXISTS (
    SELECT 1 FROM customers c
    WHERE c.id = o.customer_id AND c.active = 1
);
```

- Delta format: An open-source data storage format that adds a metadata layer on top of data files (like Apache Parquet) to bring ACID transactions, schema enforcement/validation, and data versioning to data lakes.
    - Parquet + Transaction Log (_delta_log/)
    - CDF (Change data feed): Track and read only the data that has changed
    - Optimistic concurrency with version checks; last writer wins unless conflict in case of concurrent writes in delta.
    - MERGE vs INSERT OVERWRITE: MERGE upserts, keeps history; INSERT OVERWRITE replaces, breaks time travel.
    - When data is written repeatedly in small batches (e.g., streaming writes, micro-batches, frequent MERGEs), Delta tables accumulate many small Parquet files — often a few KBs or MBs instead of hundreds of MBs. Small files problem in solved by Optimize + bin packing (compaction).

```
 # MERGE (Upsert)
deltaTable.alias("target").merge(
    source.alias("source"), "target.id = source.id"
).whenMatchedUpdate(set={"value": "source.value"}) \
 .whenNotMatchedInsert(values={"id": "source.id", "value":
"source.value"}) \
 .execute()
—----------------------------------------------
Merge vs Insert overwrite
Merge:
MERGE INTO user_data AS target
USING updates AS source
```

```
ON target.user_id = source.user_id
WHEN MATCHED THEN UPDATE SET target.city = source.city
WHEN NOT MATCHED THEN INSERT (user_id, name, city)
VALUES (source.user_id, source.name, source.city);
-> Keeps old versions → Time travel and CDF still work properly,
Only updates necessary rows

Insert:
INSERT OVERWRITE TABLE user_data SELECT * FROM new_snapshot;
-> Replaces existing data in the target scope (entire table or
partition). Breaks lineage of replaced partitions – time travel
still technically possible via older versions, but you lose
fine-grained row history and CDF continuity
—----------------------------------------------

# DELETE
deltaTable.delete("date < '2024-01-01'")

# UPDATE
deltaTable.update(condition="status = 'pending'", set={"status":
"'processed'"})

# Time Travel
df = spark.read.format("delta").option("versionAsOf",
5).load("/path")
df = spark.read.format("delta").option("timestampAsOf",
"2024-10-01").load("/path")

# Optimize data + Retain data commands
OPTIMIZE events ZORDER BY (country, date)
VACUUM events RETAIN 168 HOURS

# Enable CDF - delta.enableChangeDataFeed = true
spark.read.format("delta")
  .option("readChangeFeed", "true")
  .option("startingVersion", 2)
  .option("endingVersion", 3)
```

```
  .table("user_data")
```

Some more notes:

- When a file is expected to be read but is no longer present, a typical job would fail with an exception. Setting spark.sql.files.ignoreMissingFiles to true tells Spark to handle the FileNotFoundException or similar errors by simply ignoring the missing file. The job will continue to process the remaining files, and the resulting DataFrame will contain data only from the files that existed.
- foreach and foreachPartition:
    - Foreach: Runs function per row (use for debugging, not for I/O). Writing to db using this, can lead to millions of calls, hence avoid.
    - Foreachpartition: Runs function per partition on executor (ideal for external system writes)
    - Generally, they are used to write data to Kafka/RMQ/APIs/Tables, etc., termed as side-effects.

```
df = df.checkpoint() # Prevent recomputation if partition fails
# df.checkpoint(eager=True) # Materializes immediately
# df.checkpoint(eager=False) # Lazy (on next action)
df.rdd.foreachPartition(write_partition_to_dynamodb) # Efficient
parallel writes
```

- Spark is faster than Hadoop/MapReduce due to in-memory data processing, can spill memory to disk if required to handle large datasets, uses Catalyst optimizer, etc.
- printSchema() shows parquet metadata only, does not trigger a job. View plans in Spark: Use df.explain(True) or df.queryExecution.
- Foreach vs UDFs:
    - UDF and foreachPartition process data row by row, but they operate at completely different layers of Spark's architecture — and have very different purposes.
    - UDF:
        - Transform data inside the Spark SQL/DataFrame engine.

        - Operates on data values (columns, rows) as part of a transformation (select, withColumn, filter, etc.) on executors. The output of a UDF is a new column value, part of the logical plan.
    - Foreach:
        - Perform actions with side effects, often for external I/O. Also runs on executors, but outside Spark's SQL planner. Operates on raw data (RDD/Row objects) per partition. Used after all transformations are done.
- Photon is Databricks' next-generation vectorized execution engine, written in C++, designed to replace many of Spark's traditional JVM operators.
    - How Photon runs:
        - Spark normally executes physical operators within JVM-based executors.
        - When Photon is enabled and the query is compatible:
            - Spark's logical plan → Photon physical operators
            - Instead of running Java bytecode, executors load Photon's native C++ operators.
        - Photon uses the same columnar memory format (Arrow-like) that Spark uses → minimal copying.
    - Why C++ execution is faster
        - Python → JVM involves Py4J, IPC, serialization overhead → slow.
        - JVM → C++ (Photon) stays within the executor process using shared memory buffers → near zero serialization.
        - Photon uses: SIMD vectorization, tighter memory layout, CPU instruction-level optimizations (AVX-512, etc.)
    - Where Photon works: Photon accelerates: File Scans (Parquet, Delta), Projection, Filter, Aggregations, Joins (hash, sort-merge).
    - Where Photon does NOT work: Python / Scala UDFs, Complex nested types in some cases, MLlib operators, Graph workloads.
        - When unsupported sections appear, Spark falls back to JVM operators only for those parts—the rest of the plan can still use Photon.

—————————————————————————————————————

Structured Streaming [High-level basics]

- Core:
    - Spark treats streaming as incremental batch processing. Source → Ingestion → Transformations → Sink. DAG Scheduler handles each micro-batch as a separate job.
    - Unbounded Data: Continuous stream of incoming records (e.g., Kafka, socket, files).
    - Streaming DataFrame: Logical table that keeps growing over time.
    - Trigger: Defines when to process data (e.g., `Trigger.ProcessingTime("10 seconds")`).
    - Micro-batch Mode (default): Spark batches incoming data into small chunks for processing.
    - Continuous Mode (experimental): Processes data record-by-record for sub-second latency (rarely used in prod).
    - Trigger.Once: Runs one micro-batch and stops (useful for incremental ETL).
- Checkpointing: Essential for fault tolerance and recovery. Stores (in HDFS/S3): Offsets of data source (e.g., Kafka offset), State store snapshots (for aggregations/joins), Metadata for job progress. Without checkpointing, Spark can't resume from failure.
- Watermarking: Tells Spark how long to wait for late data. Example: withWatermark("event_time", "10 minutes"). Late data is discarded.
- Operations:
    - Stateless: Eg: filter, map, select; Does not use any statestore (HDFS/S3).
    - Stateful: Eg: aggregations, windowed operations, joins; Uses statestore.
- Stream joins:
    - Stream–Static Join: stream joined with a static lookup table
    - Stream–Stream Join: Both sides unbounded → requires watermark + state handling

- Output Modes:
    - Append: Only new rows (e.g., no aggregation)
    - Update: Update existing aggregations incrementally
    - Complete: Recompute entire output table every batch (for full aggregates)
- Sink:
    - In-memory table for testing / Spark SQL queries
    - File (Parquet/JSON) writes in micro-batches
    - Kafka; Exactly-once supported
    - foreach: Row-level custom sink. Risky — not idempotent
        - It's not idempotent by default. If a micro-batch fails halfway through and restarts, Spark reprocesses that batch from the last checkpoint. foreach directly calls user logic (e.g., insert into DB, PUT to S3). Spark's checkpoint only tracks offsets processed, not what your writer did externally. Runs directly on workers.
    - foreachBatch: Custom logic per batch-level. Best for DB writes (production level)
        - df.writeStream.foreachBatch(lambda batch_df, batch_id: ...): Runs once per micro-batch (not per row). BatchId is provided → you can make your logic idempotent, say writing in append mode.
        - foreachBatch is considered safe for exactly-once sinks while foreach is not.
        - The callback (the control function) runs on the driver, But the data processing inside (batch_df operations) runs distributed on executors, just like any normal DataFrame job.
- spark.streaming.backpressure.enabled=true: It adjusts the ingestion rate of data from the source.

—————————————————————————————————————

Hadoop [High-level basics]

- HDFS (Hadoop Distributed File System — Distributed Storage Layer)

    - Purpose: Reliable, fault-tolerant storage for very large files across a distributed cluster.
    - Architecture:
        - NameNode (Master): Stores metadata — file names, block locations, permissions. Keeps a transaction log (EditLog) and a checkpoint (FsImage). Doesn't store actual data, only where blocks live. Single point of failure in early Hadoop versions (fixed later via Secondary NameNode / Standby NameNode / High Availability mode).
        - DataNodes (Slaves): Store actual file blocks (default size: 128 MB or 256 MB). Send periodic heartbeats and block reports to the NameNode. Perform replication, deletion, and block creation as instructed by the NameNode.
    - Key Concepts:
        - Replication factor: Default = 3 → improves fault tolerance and availability.
        - Rack Awareness: NameNode knows rack topology → places replicas on different racks to balance network efficiency and fault resilience.
        - Write Path: Client → NameNode (for metadata) → Pipeline to DataNodes for block storage.
        - Read Path: Client contacts NameNode → retrieves block locations → reads directly from nearest DataNodes.
    - Benefits: High throughput, scalable storage, fault-tolerant, optimized for large sequential reads/writes.
- MapReduce (Distributed Processing Framework)
    - Purpose: Parallel computation model that processes massive data using Map() and Reduce() functions.
    - Flow:
        - Job Submission: Client submits job to JobTracker (Hadoop v1) or YARN ResourceManager (v2).
        - Split Phase: Input is divided into chunks (InputSplits), typically aligned with HDFS blocks.
        - Map Phase: Mapper processes each split → outputs intermediate key-value pairs.

        - Shuffle & Sort: Framework groups all values by key and transfers them to the appropriate reducer.
        - Reduce Phase: Reducer aggregates or combines values per key → produces final output.
    - Key Points:
        - Idempotent: Multiple runs on same data yield same output.
        - Fault Tolerance: If a node fails, tasks are re-run on another node using data replicas.
        - Data Locality Optimization: Tries to schedule Map tasks on nodes where data already resides.
        - Output written back to HDFS.
    - Limitation: High latency for iterative or real-time processing — hence, frameworks like Spark were built to overcome this.
- YARN (Yet Another Resource Negotiator — Cluster Resource Management Layer)
    - Purpose: Decouples resource management from job scheduling, making Hadoop cluster multi-tenant and more efficient.

>>

Scenario(s)

Question1: Assume daily data volume to be 1TB/day from Kafka. Discuss about infra requirements if you were to ingest/parse data and store in S3 - in daily basis (batch) or near real-time basis (streaming).

Answer:

Common Assumptions

Parameter	Value	Notes
Daily data volume	1 TB/day (≈ 1000 GB)	Uncompressed JSON input

Average record size	1 KB	≈ 1 billion records/day
Storage layer	HDFS / S3 / Delta Lake	
Compression after parse	Parquet (≈ 5× smaller)	
Cluster type	Spark on YARN or Kubernetes	
Task	JSON parsing → schema mapping → write Parquet/Delta	

Batch Job (Run Once per Day):

- Workload Pattern
  - One Spark job runs once a day.
  - Reads 1 TB JSON → parses → writes to Parquet.
  - Target: finish in ~1 hour.
- JSON Parsing Throughput
  - Average: 30 – 50 MB/sec per vCPU.
  - Assume 40 MB/sec/vCPU for estimation.
  - Formula: Total time (sec) = (1000 GB × 1024 MB/GB) / (Throughput × vCPUs)
  - Target time = 3600 sec → vCPUs ≈ (1000 × 1024) / (40 × 3600) ≈ 7 cores (ideal)
  - Add ~5× overhead for Spark shuffles, GC, I/O, etc. → ≈ 40–50 vCPUs needed realistically.
- Suggested Infra (Batch): Note that if you're fine with 2–3 hr runtime, halve the cores (~20–25 vCPUs).

Resource	Estimate

Executors	10 – 12
Cores / executor	4 – 5
Total vCPUs	40 – 50
Memory / executor	16 – 24 GB
Total memory	200 – 250 GB
Cluster nodes	~5–6 × m5.4xlarge (AWS)
Cost	~ $3–4 per hour of runtime

Near Real-time Streaming Job:

- Workload Pattern
  - Continuous Kafka ingestion rate: 1 TB / 24 h = 1000 GB / 86400 s ≈ 11.6 MB/sec
  - Goal: process each micro-batch within seconds or a minute.
- Efficiency
  - Structured Streaming is ~2–3× less efficient than batch due to micro-batching and checkpointing.
  - Rule of thumb: ~2 vCPUs per MB/sec input rate
  - At 11.6 MB/sec input: 11.6 × 2 vCPUs ≈ 23 vCPUs
  - Add 50 % headroom → ≈ 35–40 vCPUs total (continuous).
- Suggested Infra (Streaming): Note that streaming runs 24×7, unlike batch.

Resource	Estimate

Executors	8 – 10
Cores / executor	4
Total vCPUs	32 – 40
Memory / executor	16 – 24 GB
Total memory	160 – 200 GB
Cluster nodes	~4–5 × m5.4xlarge (AWS)
Cost	$3–4/hr × 24 h = $75–100/day

Comparison Summary

Aspect	Batch (Daily)	Streaming (Near Real-time)
Data processed	1 TB once/day	Continuous 1 TB/day
Compute needed	~40–50 vCPUs (1 hr run)	~35–40 vCPUs continuous
Memory	~200–250 GB	~160–200 GB
Cost	~$3–4/hr (only 1 hr/day)	~$75–100/day

Latency	Up to 1 day	Seconds – minutes
Complexity	Easier	Higher (checkpointing, watermarking)

Key Takeaways

- Batch = cheaper, simpler if 1-day delay acceptable. Streaming = real-time insights but higher cost (~20–30× more).
- JSON parsing is CPU-intensive — prefer Avro or Parquet ingestion.
- Autoscaling or serverless Spark (Databricks, EMR on EKS) reduces idle cost.
- Can I use less infra for streaming to save cost?
    - Yes — if your streaming job does very little per-record work, one or two vCPUs may keep up with a modest Kafka throughput. But if your streaming job performs CPU-heavy JSON parsing, joins, windowing, stateful operations, or frequent checkpoints, you need many more cores to avoid lag and operational issues.
    - Trade-offs when you reduce infra
        - Increased processing latency — microbatches take longer to finish → higher end-to-end latency (seconds → minutes → hours).
        - Backpressure and Kafka lag — consumer lag will grow if processing rate < input rate; lag must be stored in Kafka (requires sufficient retention).
        - Larger state / checkpoint growth — slower processing increases state growth and checkpoint sizes, slowing recoveries.
        - Spill to disk / GC pressure — insufficient memory leads to disk spill and longer task times; long GC pauses may cause executor failures.
        - Lower fault tolerance and recovery speed — on failure, recovery needs more time if checkpoints are large and cluster is small.

        - Higher operational complexity — must monitor lag, tune partitions, tune microbatch durations, autoscale carefully.
        - Potential for data loss if retention insufficient — if Kafka retention expires before backlog is processed.
    - Example scenarios (for 1 TB/day → 11.6 MB/s)
        - Scenario A — Light work (minimal parsing):
            - per_core_MBps = 12 MB/s
            - required_cores = 11.6 / 12 ≈ 1.0
            - headroom 2× → allocated_cores ≈ 2 vCPUs. When appropriate: messages are already compact (AVRO/Protobuf), transforms are trivial, no stateful operations.
        - Scenario B — Medium work (JSON parsing + enrichments)
            - per_core_MBps = 4 MB/s
            - required_cores = 11.6 / 4 ≈ 2.9
            - headroom 2× → allocated_cores ≈ 6 vCPUs
              When appropriate: parsing JSON, some UDFs, light joins, small state windows.
        - Scenario C — Heavy work (stateful joins, large windows, heavy UDFs)
            - per_core_MBps = 1 MB/s
            - required_cores = 11.6 / 1 ≈ 11.6
            - headroom 2× → allocated_cores ≈ 24 vCPUs
              When appropriate: large keyed state, big aggregations, frequent checkpoints, complex UDFs.
        - Note: memory per executor should be aligned to avoid spills (e.g., 16–32 GB per executor depending on JVM overhead and state size). Also distribute workload across many Kafka partitions to parallelize.

>>

Question2: -

Answer:-

>>>