Breaking Data Barriers: Transforming ETL with Apache Spark

Sudhir Nikam | sudhirnikam.me

What we will discuss today?

OLTP vs OLAP
ETL vs ELT
Data Barriers - Explanation
What is Apache Spark
Apache Spark Architecture
Data Barriers - Solution
Apache Spark Internals
Demo

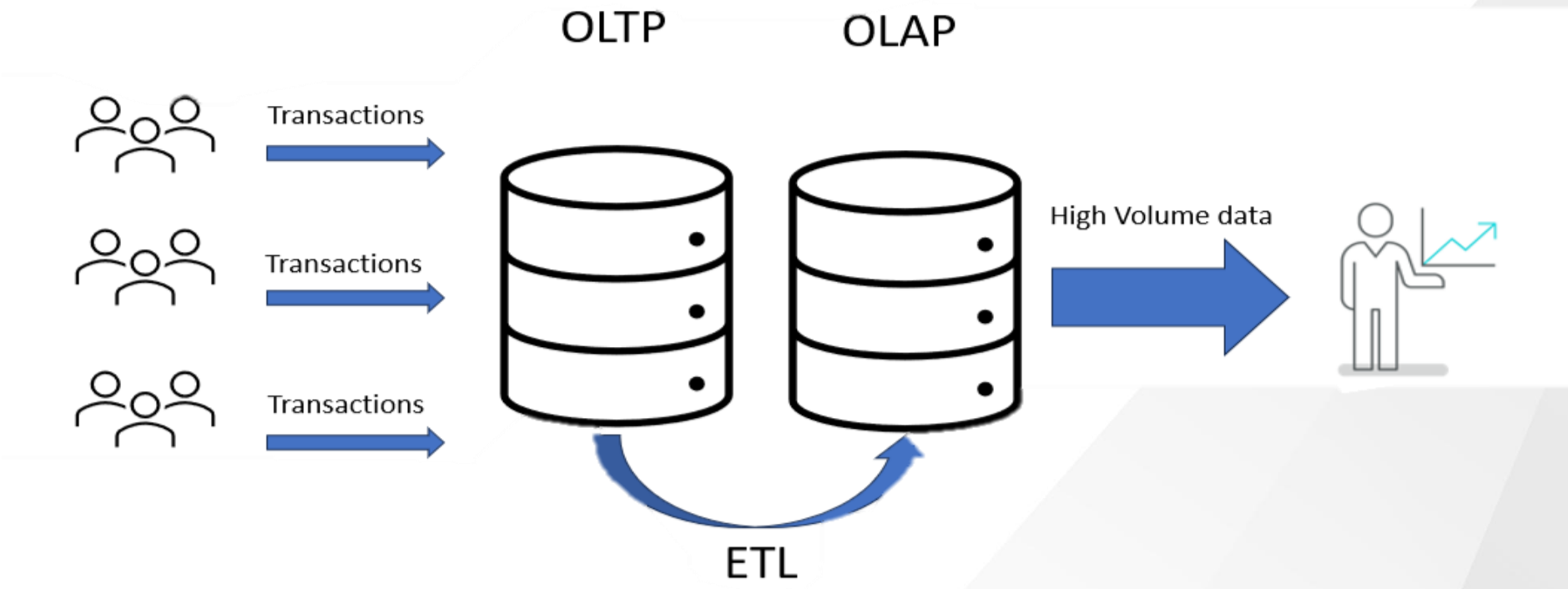
What is out of scope?

Deep Spark internals (RDD APIs, Custom Schedulers)
Advanced tuning (AQE, Broadcast Hints, Tungsten)
Production troubleshooting & debugging
ML libraries (MLlib, Spark ML pipelines)

OLTP vs OLAP

OLTP	OLAP
Online Transaction Processing	Online Analytical Processing
Runs the business by day	Reports on the business by night
Handles day-to-day operations - inserts, updates, deletes	Handles complex queries for business intelligence
Optimized for fast, concurrent transactions	Optimized for read-heavy analytical workloads
Normalized schema to avoid redundancy	Denormalized schema for query performance
Example: Banking transactions, order processing, inventory management	Example: ETL pipelines, Sales analytics, trend analysis, data warehousing, ML training

OLTP to OLAP Architecture



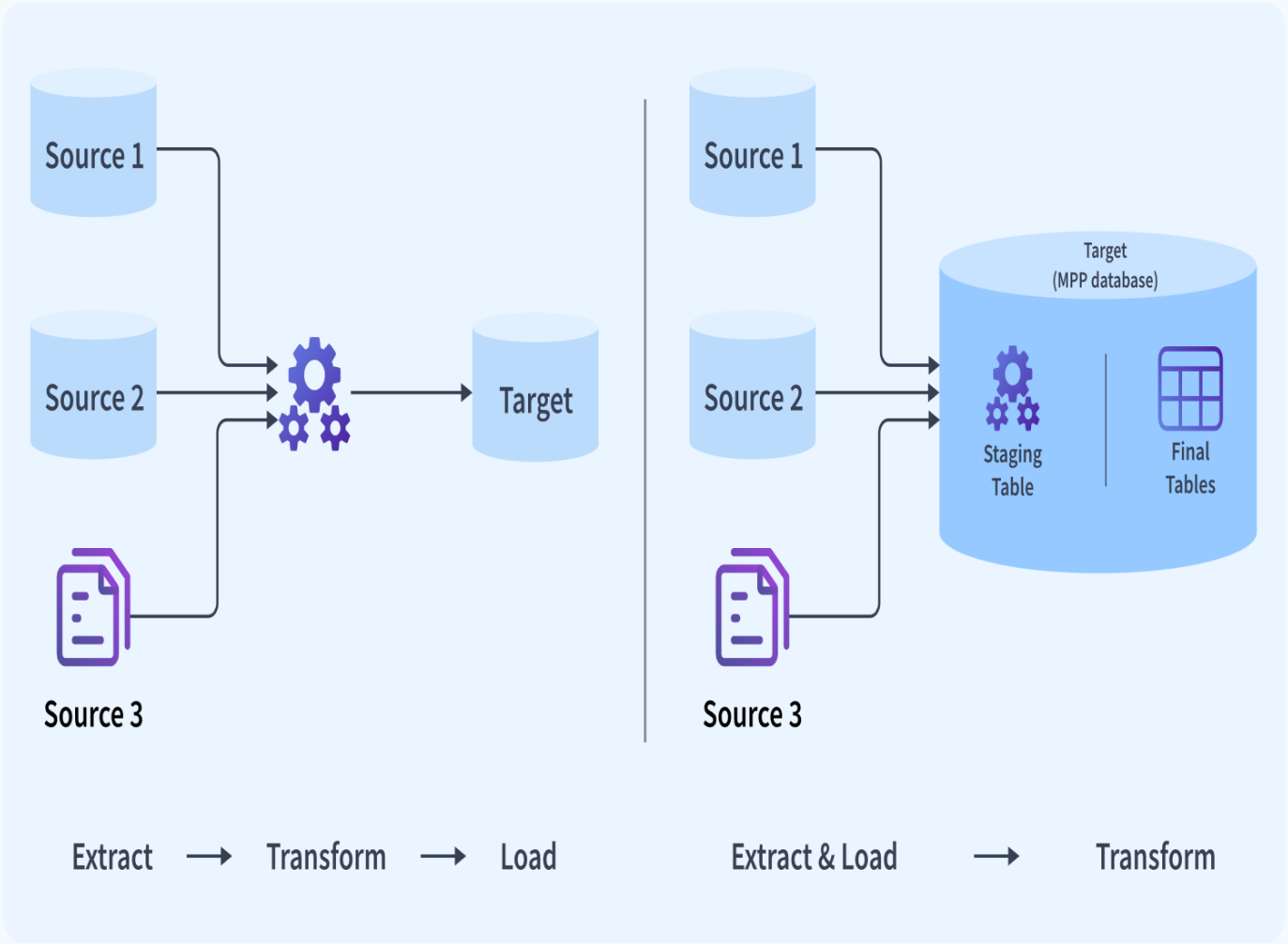
ETL vs ELT

ETL:

- You **extract** raw data from various sources
- You use a secondary processing server to **transform** that data
- You **load** that data into a target database

ELT:

- You **extract** raw data from various sources
- You **load** it in its natural state into a data warehouse or data lake
- You **transform** it as needed while in the target system



Data Barriers(OLAP) - The Reality

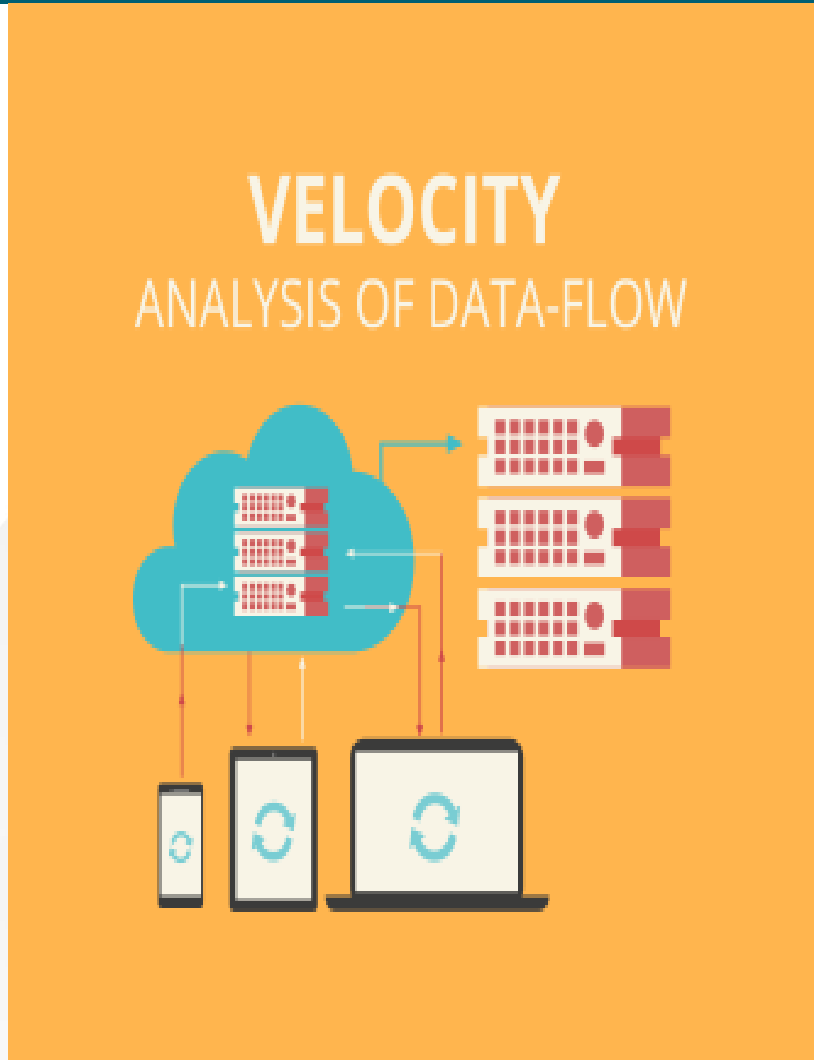
- Volume
- Velocity
- Variety
- Veracity(Accuracy or Truthfulness)
- No Single Source of Truth
- Legacy ETL Tools

Volume - Petabytes of Data

- Traditional tools built for gigabytes now struggle with terabytes and petabytes from transactions, IoT sensors, logs, and user behavior data
- Business Impact:
  - Processing delays
  - Infrastructure costs
  - Missed opportunities
- Business example: A **large volume of financial data** is stored in an **Oracle DB**, and legacy tools are too slow to process it efficiently

Velocity - Real-time Processing

- Business events happen in milliseconds, but traditional ETL takes hours or days—missing the window for real-time action
- Business Impact:
  - Competitive disadvantage
  - Revenue loss
  - Customer dissatisfaction
- Business example: Traditional ETL platforms are optimized for batch workloads and **lack the capability to process real-time streaming** data from systems such as **Kafka**

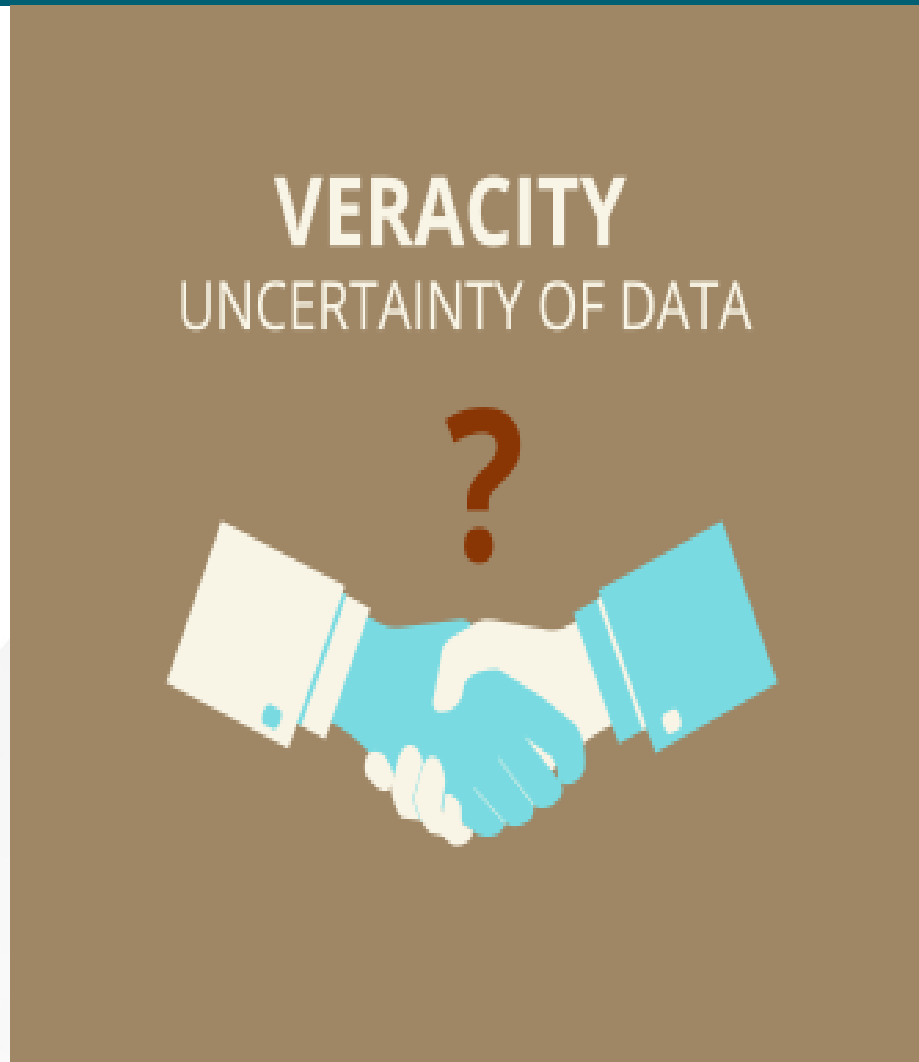


Variety – Multiple Source Formats

- Data spans multiple formats—APIs, JSON, CSV, Parquet, XML, Streams, COBOL
- Business Impact:
  - Analysis paralysis
  - Incomplete view
- Business example: The data team receives inputs from a wide range of sources, including CSV, Fixed Width, **Fixed Width Multiline Text**, Excel, RDBMS systems, **COBOL files**, JSON, and **various NoSQL databases**

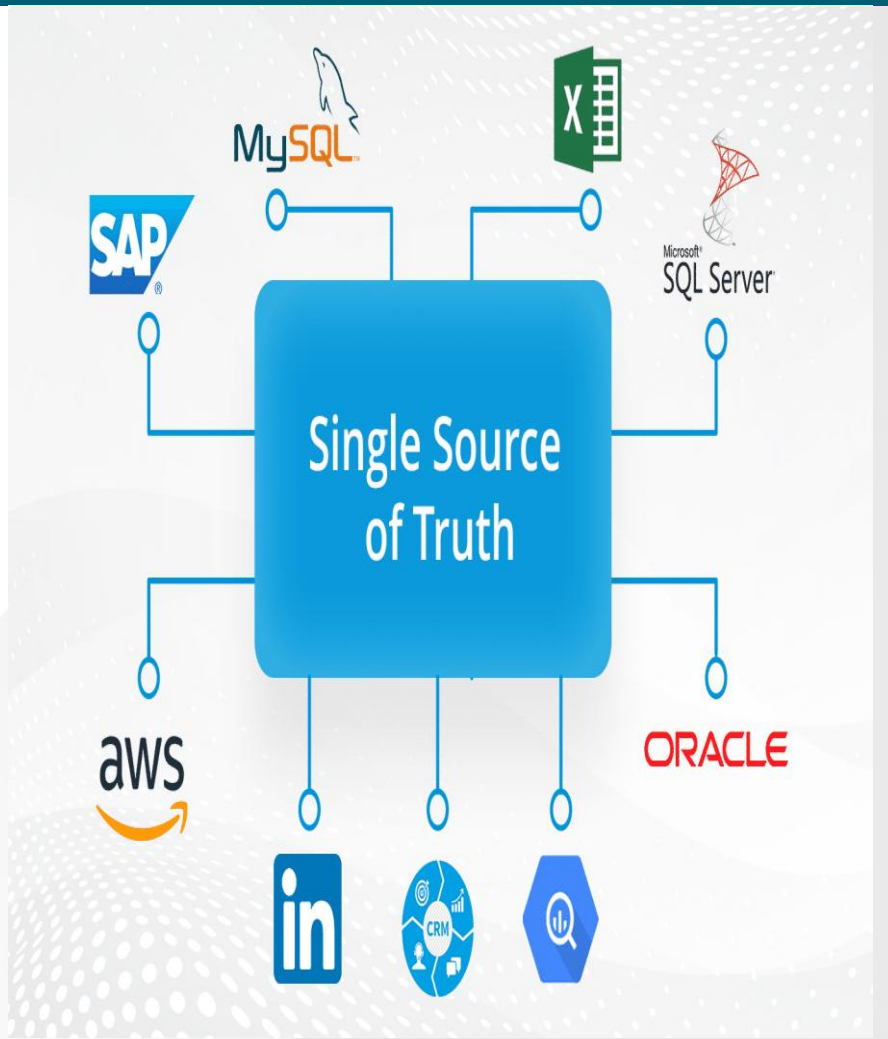
Veracity: Ensuring Data Accuracy and Reliability

- Duplicate records, missing values, conflicting formats, and outdated information create unreliable datasets across systems
- Business Impact:
  - Wrong decisions
  - Compliance risks
  - Customer trust erosion
- Business example: Critical fields such as **date of birth**, **Social Security Number**, or **address** are missing from the dataset



No Single Source of Truth

- Critical data scattered across CRM, ERP, Data Warehouses, and Cloud storage—each system maintaining its own truth
- Business Impact:
  - Conflicting reports
  - Decision paralysis
  - Operational inefficiency
- Business example: Oracle and Kafka **contain incomplete and inconsistent data**, making reliable reconstruction impossible



Legacy ETL Tools

- Traditional ETL tools designed for batch processing fail at scale—requiring expensive licenses, expertise, and inflexible pipelines
- Business Impact:
  - Scalability ceiling
  - Time-to-insight delay
  - Cost explosion
- Business example: **High licensing costs** associated with legacy ETL tools

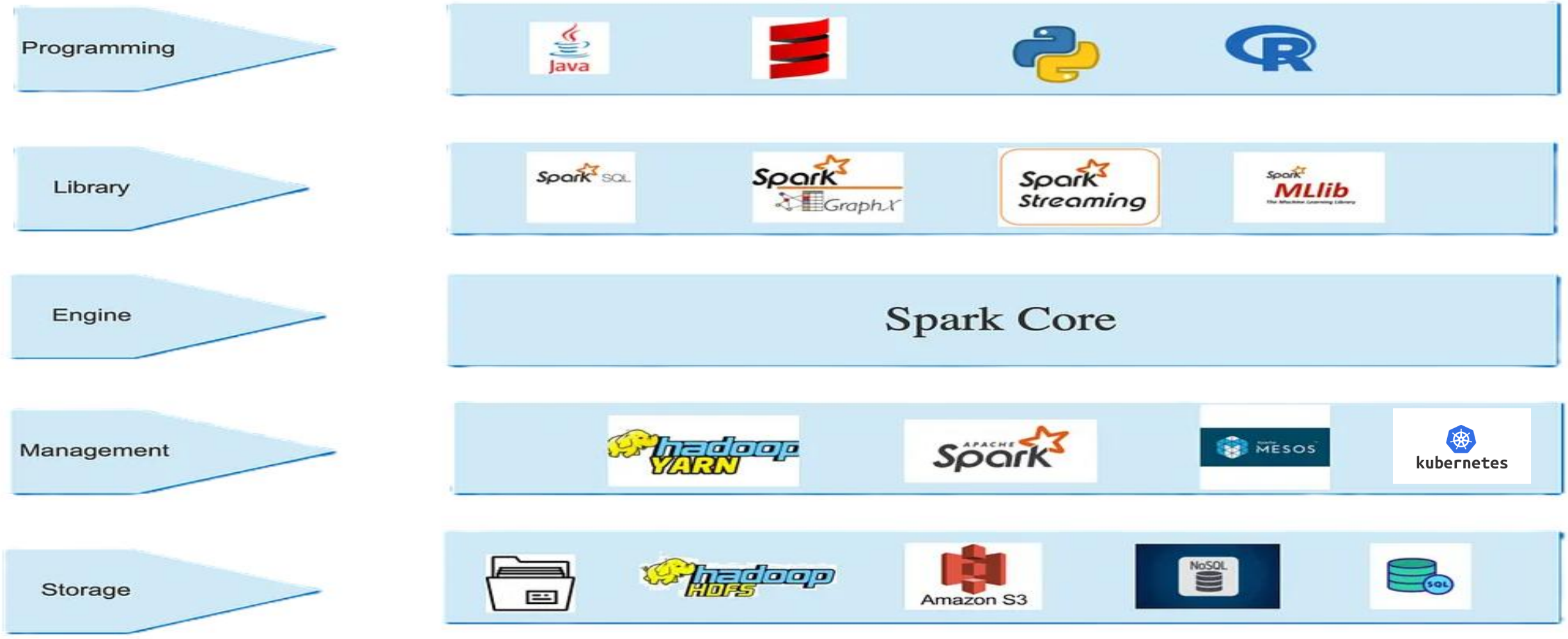
Hidden Cost

Bad Data = Bad Decisions = Business Failure

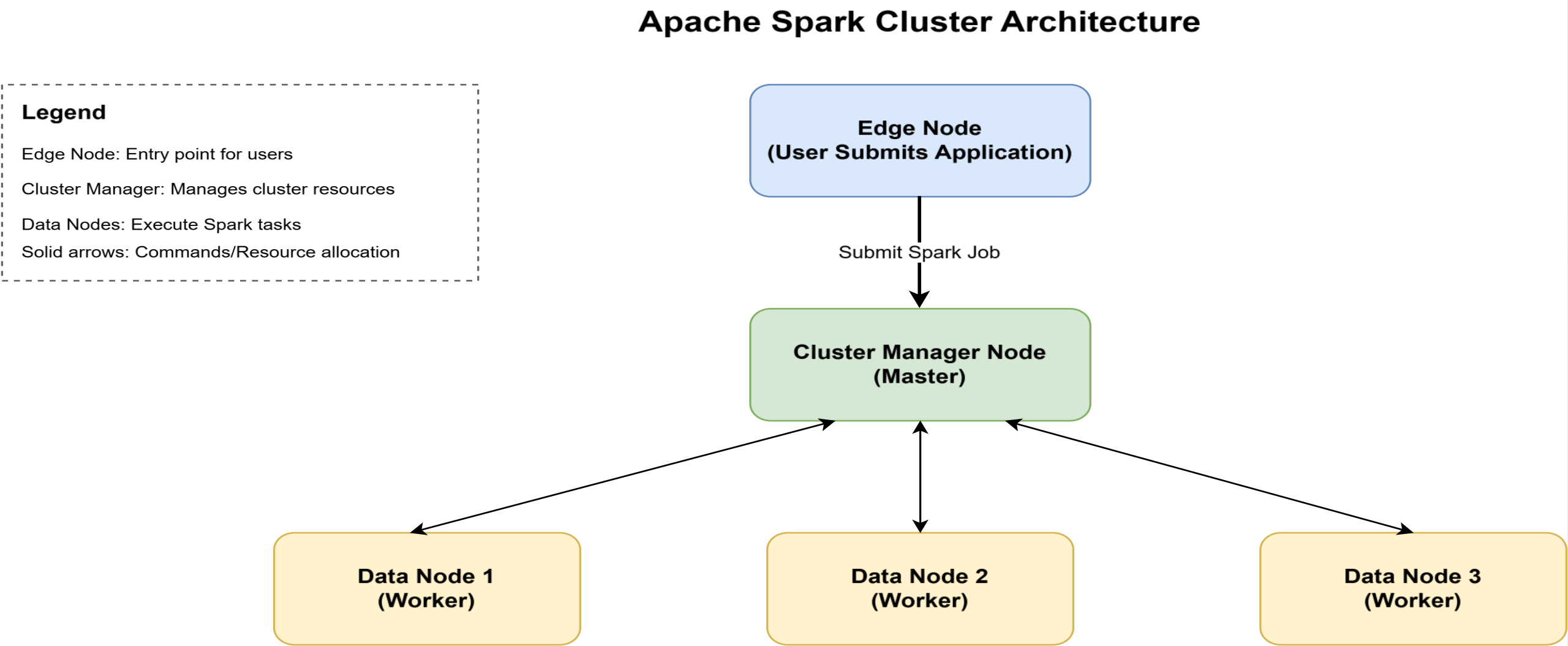
What is Apache Spark?

- Open-source distributed computing framework that processes massive amounts of data at lightning speed by distributing work across clusters of computers
- The Spark Advantage:
  - ⚡ 100x faster than traditional MapReduce (in-memory processing)
  - 🔄 Unified platform - Batch, Streaming, SQL, ML in ONE engine
  - 📈 Linear scalability - From gigabytes to petabytes seamlessly
  - 💰 Open source - Zero licensing costs, no vendor lock-in
  - 🐍 Easy to use - Write in Python, Scala, Java or SQL

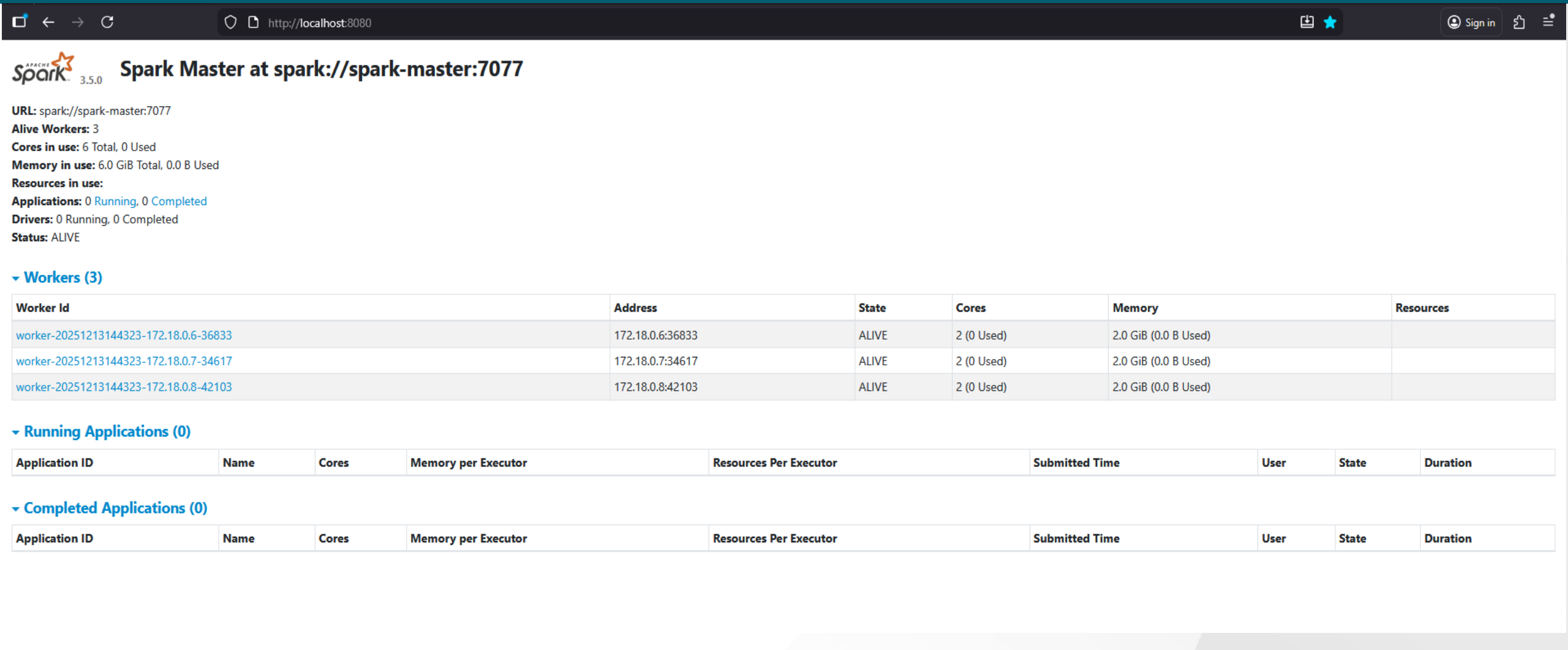
Spark Ecosystem



Apache Spark Architecture



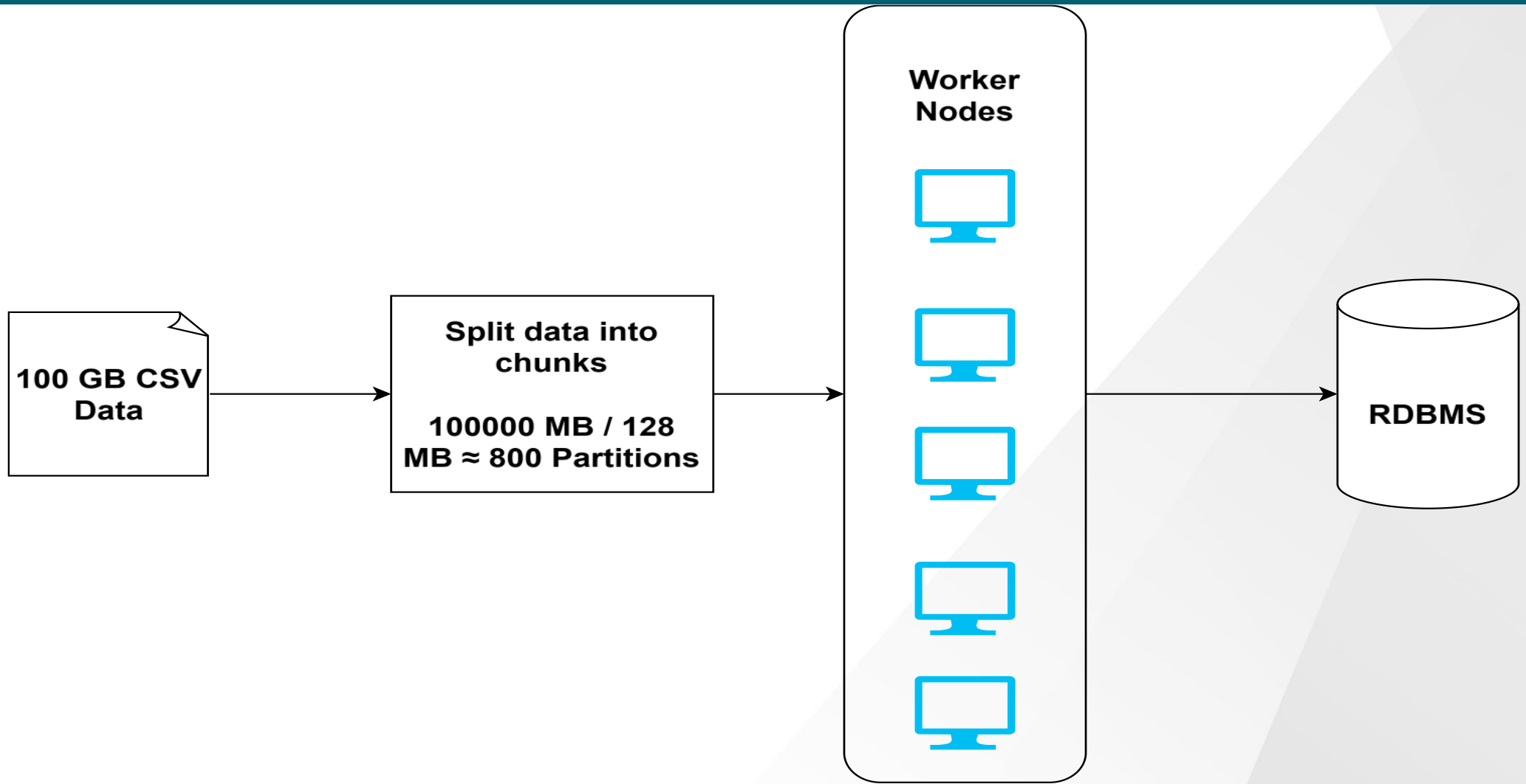
Apache Spark Cluster – Standalone Mode



Solving Volume - Distributed Processing

- How Spark Solves It:
  - Horizontal scaling across 100s-1000s of commodity servers
  - In-memory computing: 100x faster than disk-based processing
  - Intelligent partitioning distributes data evenly across cluster
  - Lazy evaluation optimizes execution before running
- Business Outcome:
  - Batch jobs runtime → reduced from hours to mins
  - Significant reduction in infrastructure costs
  - Analyze complete datasets, not samples

Solving Volume – Spark Way



Solving Velocity - Real-Time Streaming

- How Spark Solves It:
  - Micro-batch processing with sub-second latency
  - Structured Streaming: same API for batch and real-time data
  - Exactly-once processing guarantees
  - Native integration with Kafka, Kinesis
- Business Outcome:
  - Real-time fraud detection in milliseconds
  - Dynamic pricing updates every 5 minutes
  - Live dashboards with <1 minute latency

Solving Velocity – Spark Way

Source Type	Code Snippet
Kafka	df = spark.readStream.format("kafka").option("kafka.bootstrap.servers", "host:9092").option("subscribe", "topic_name").load()
AWS Kinesis	df = spark.readStream.format("kinesis").option("streamName", "stream_name").option("region", "us-east-1").option("initialPosition", "latest").load()
Azure Event Hubs	df = spark.readStream.format("eventhubs").option("eventhubs.connectionString", "connection_string").load()
RabbitMQ (using third-party connector)	df = spark.readStream.format("rabbitmq").option("host", "host").option("queue", "queue_name").load()
Socket Stream (for testing/demos)	df = spark.readStream.format("socket").option("host", "localhost").option("port", 9999).load()

Solving Variety - Unified Data Processing

- How Spark Solves It:
  - Native support: JSON, Parquet, CSV, Avro, ORC, XML, Delta
  - Schema-on-read
  - Single DataFrame API for all data types
- Business Outcome:
  - Data integration: weeks → hours
  - 70% time on transformation → 70% time on analysis
  - Faster time-to-insight

Solving Variety – Spark Way

Source Type	Code Snippet
CSV	df = spark.read.csv("path/to/file.csv", header=True, inferSchema=True)
JSON	df = spark.read.json("path/to/file.json")
COBOL(Mainframe)	df = spark.read.format("cobol").option("copybook", "path/to/copybook.cpy").load("path/to/data")
Parquet	df = spark.read.parquet("path/to/file.parquet")
RDBMS (JDBC)	df = spark.read.jdbc(url="jdbc:postgresql://host:5432/db", table="table_name", properties={"user": "user", "password": "pass"})
NoSQL - MongoDB	df = spark.read.format("mongo").option("uri", "mongodb://host:27017/db.collection").load()

Solving Veracity - Data Quality at Scale

- How Spark Solves It:
  - Built-in deduplication: dropDuplicates() across billions of rows
  - Null handling: fillna(), dropna(), coalesce()
  - Custom validation with UDFs and business rules
  - Window functions for anomaly detection
- Business Outcome:
  - Improved Data quality
  - Automated validation prevents bad data in reports
  - Consistent definitions across all systems

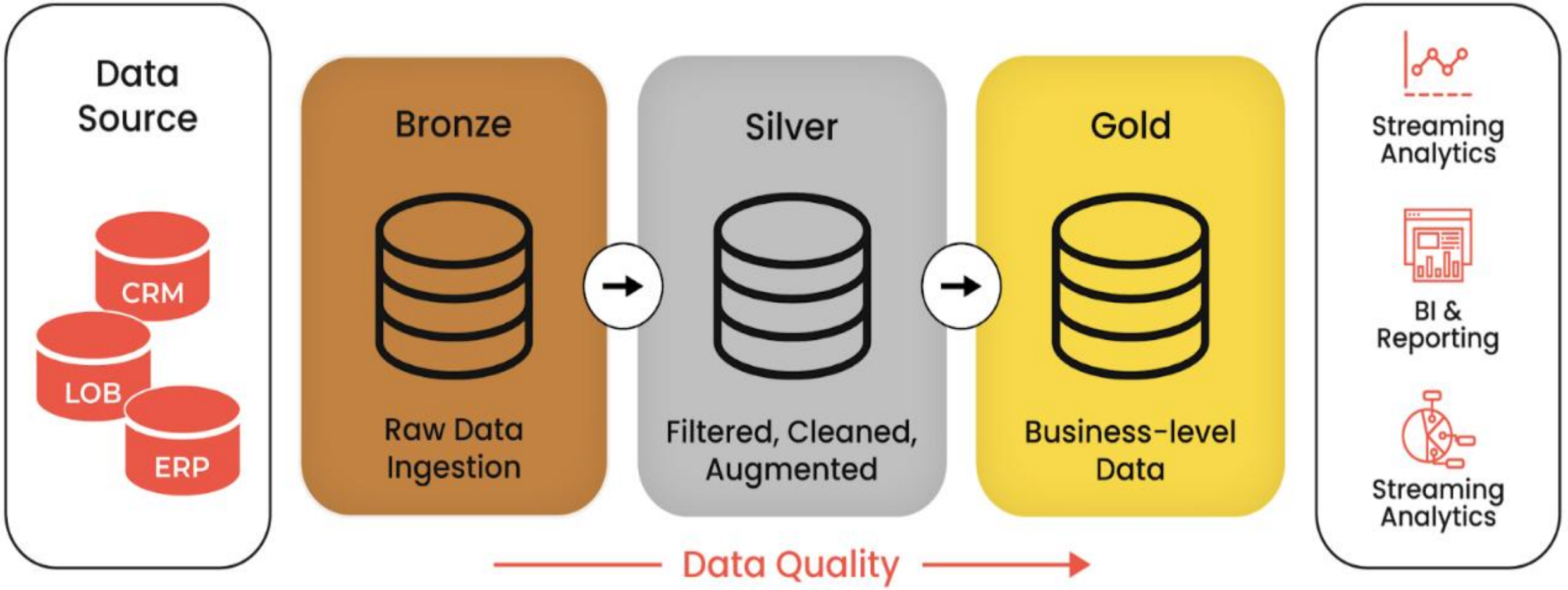
Solving Veracity – Spark Way

Data Quality Check	Code Snippet
Remove null values	df_clean = df.dropna(subset=["critical_column"])
Remove duplicates	df_clean = df.dropDuplicates(["id", "email"])
Filter invalid records	df_valid = df.filter((col("age") > 0) & (col("age") < 120) & col("email").contains("@"))
Standardize data	df_clean = df.withColumn("email", trim(lower(regexp_replace(col("email"), "[^a-zA-Z0-9@.]", ""))))
Schema validation with enforcement	df_clean = spark.read.schema(expected_schema).option("mode", "DROPMALFORMED").csv("path/to/file.csv")
Complex multi-condition validation	df_valid = df.filter((col("amount") > 0) & (col("date") <= current_date()) & col("status").isin(["ACTIVE", "PENDING"]))

Solving Single Source of Truth

- How To Solves It:
  - Bronze Layer
  - Silver Layer
  - Gold Layer
- Business Outcome:
  - All teams query the same validated data
  - No more "whose numbers are correct?" debates
  - Business users access governed, trusted data

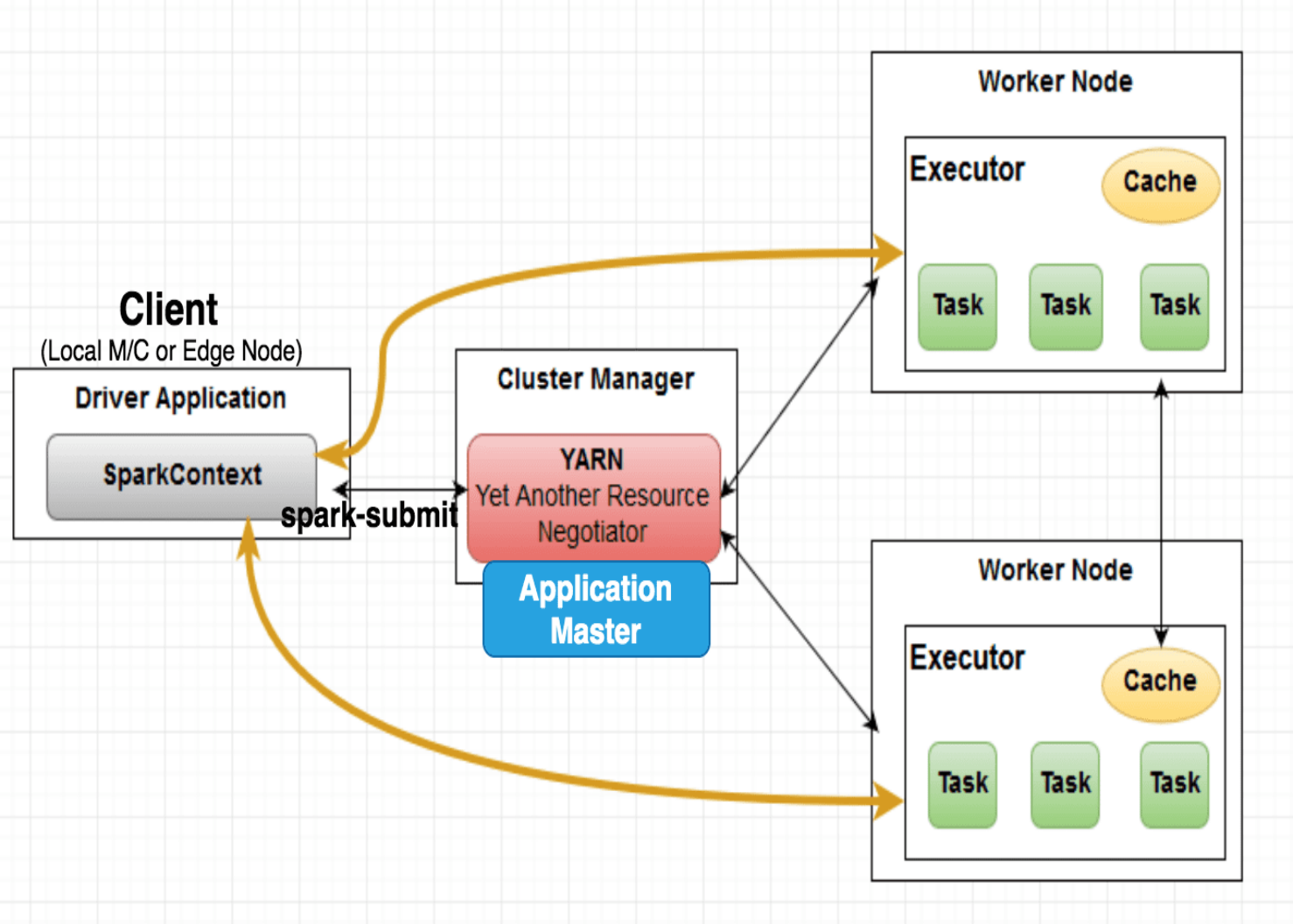
Medallion Architecture



Spark Application Submit

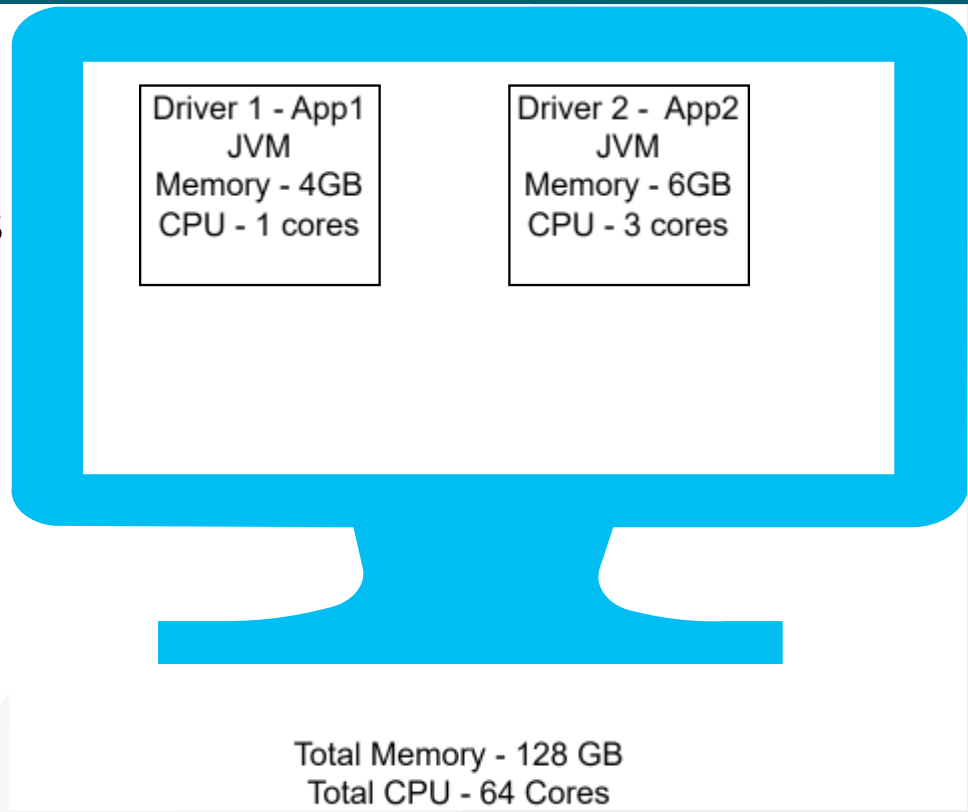
```
spark-submit \
  --master yarn \
  --deploy-mode client \
  --driver-memory 3g \
  --driver-cores 2 \
  --executor-memory 4g \
  --executor-cores 4 \
  --num-executors 2 \
  --py-files dependencies.zip \
  my_app.py
```

--master k8s://https://<kube-apiserver>:6443
--master mesos://mesos-master:5050
--master spark://spark-master-host:7077



What is Driver?

- Runs the main() function
- Analyzes user code, builds the DAG, and divides work into tasks
- Schedules and coordinates task execution across executor nodes
- Returns final results to the user
- One Driver Per Spark Application

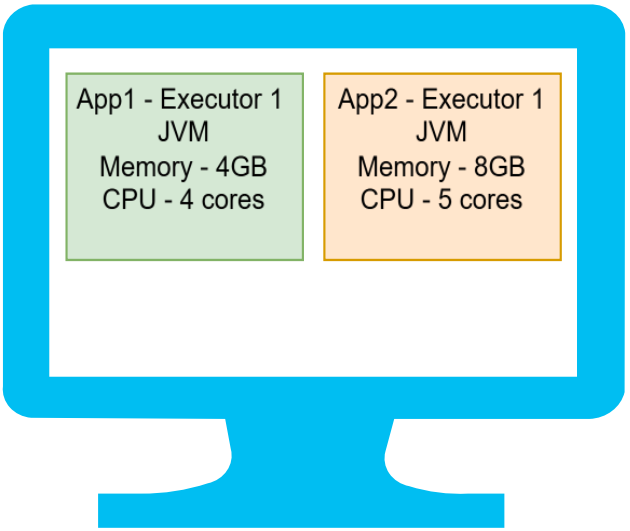


What is Executor?

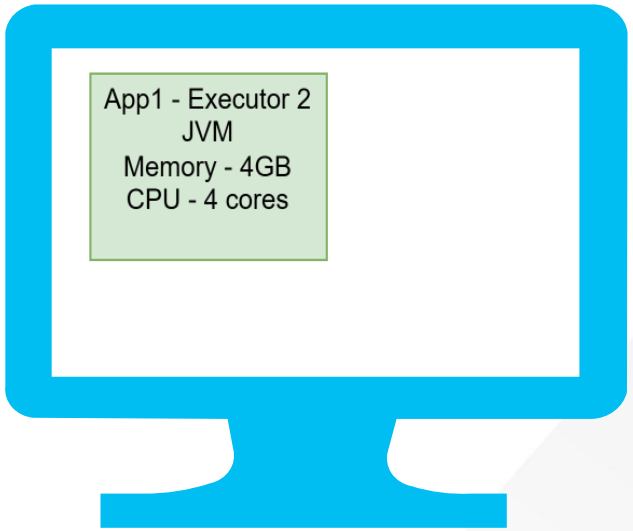
- Distributed worker process running on cluster nodes
- Executes tasks assigned by the Driver in parallel
- Caches data partitions in memory for fast access
- Runs for the entire application lifetime with dedicated CPU and memory
- More than one executors per Spark Application

Spark Executors

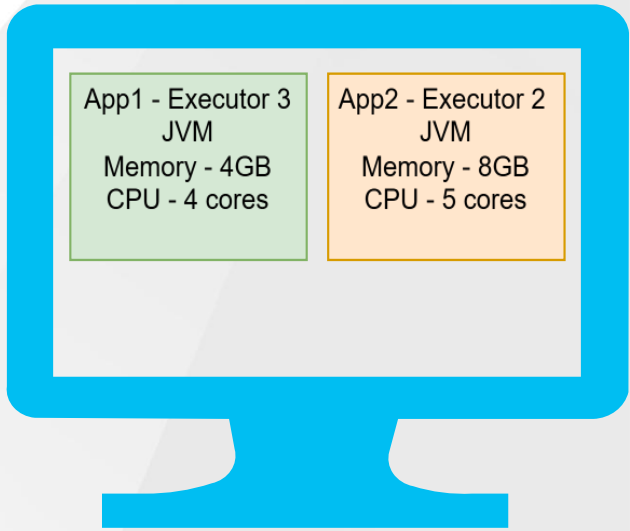
Spark Application 1 - Requested 3 executors, each with 4GB memory and 4 CPU cores
Spark Application 2 - Requested 2 executors, each with 8GB memory and 5 CPU cores



Worker Node - 1
Total Memory - 128 GB
Total CPU - 64 Cores

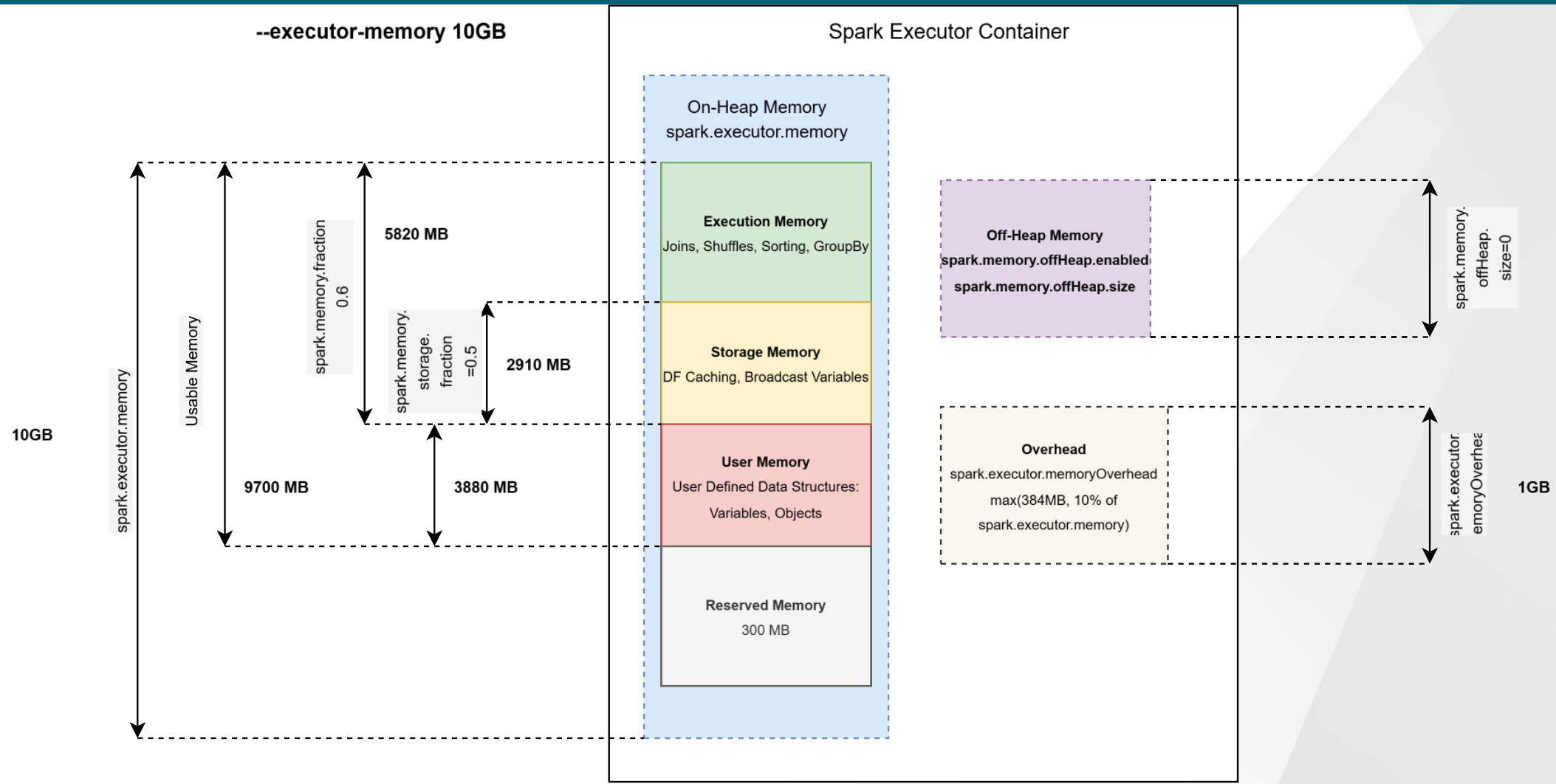


Worker Node - 2
Total Memory - 128 GB
Total CPU - 64 Cores



Worker Node - 3
Total Memory - 128 GB
Total CPU - 64 Cores

Executor Container Memory Management



Spark Dataframe

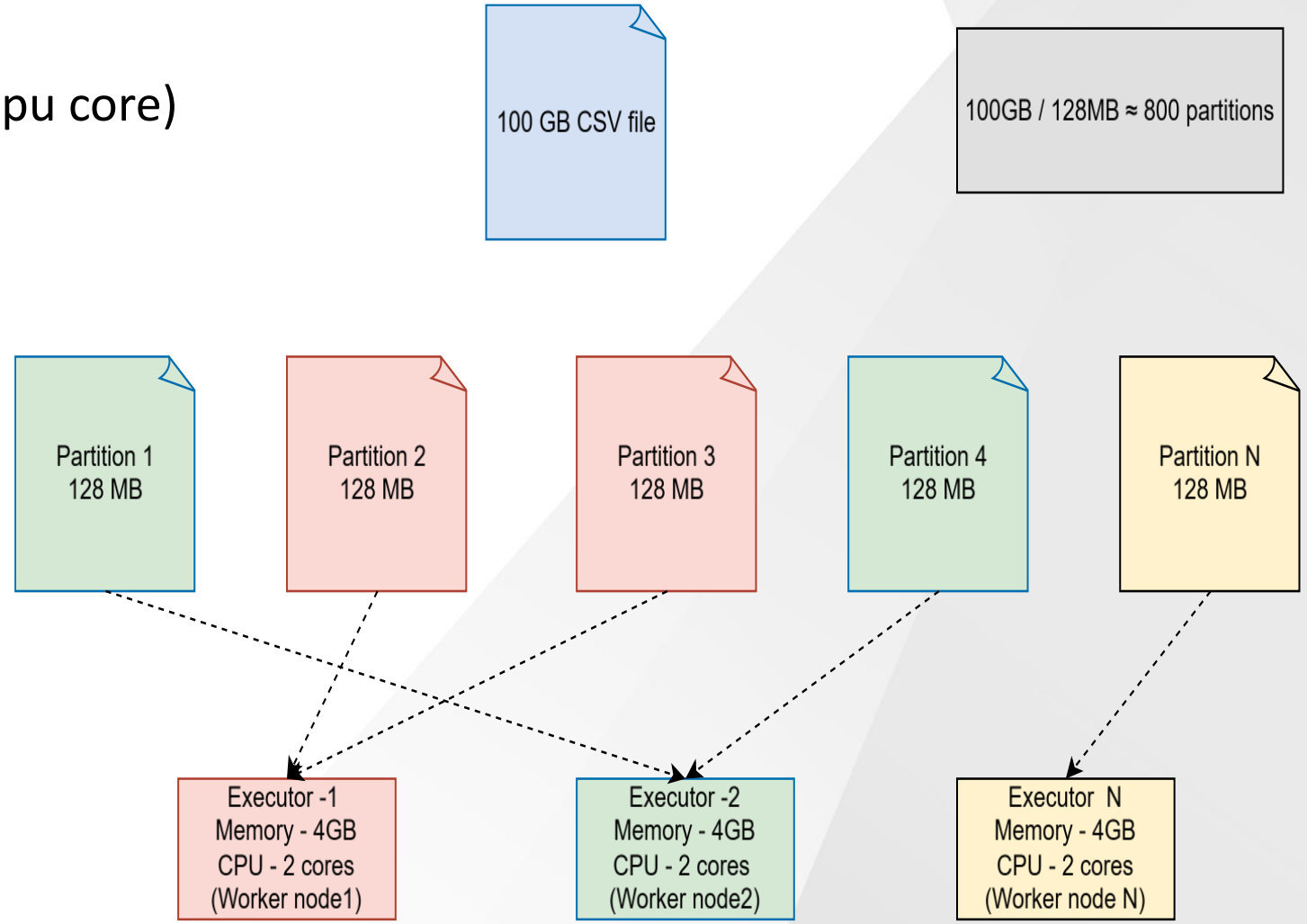
- Distributed collection of data organized in named columns(**in memory**)
- Like a table in a database, but distributed across cluster
- Immutable - transformations create new DataFrames
- Foundation for structured data processing in Spark

```python
# Define schema for transactions
schema = StructType([
    StructField("id", LongType(), False),
    StructField("date", StringType(), False),  # Will convert to timestamp
    StructField("client_id", IntegerType(), False),
    StructField("card_id", IntegerType(), False),
    StructField("amount", StringType(), False),  # Read as string first
    StructField("use_chip", StringType(), False),
    StructField("merchant_id", IntegerType(), False),
    StructField("merchant_city", StringType(), True),
    StructField("merchant_state", StringType(), True),
    StructField("zip", StringType(), True),
    StructField("mcc", IntegerType(), True),
    StructField("errors", StringType(), True)
])

# Read CSV from MinIO, file size transactions_data.csv = 1.17GB
print("Reading data from MinIO...")
df = spark.read \
    .option("header", "true") \
    .schema(schema) \
    .csv("s3a://dev/transactions_data.csv")

print(f"Number of partitions: {df.rdd.getNumPartitions()}")
print("Schema:", df.dtypes)
df.show(5, False)
```

```
Reading data from MinIO...
Number of partitions: 10
Schema: [('id', 'bigint'), ('date', 'string'), ('client_id', 'int'), ('card_id', 'int'), ('amount', 'string'), ('use_chip', 'string'), ('merchant_id', 'int'), ('merchant_cit
y', 'string'), ('merchant_state', 'string'), ('zip', 'string'), ('mcc', 'int'), ('errors', 'string')]
+-------+-------------------+---------+-------+-------+----------------+-----------+-------------+--------------+-------+----+------+
|id     |date               |client_id|card_id|amount |use_chip        |merchant_id|merchant_city|merchant_state|zip    |mcc |errors|
+-------+-------------------+---------+-------+-------+----------------+-----------+-------------+--------------+-------+----+------+
|7475327|2010-01-01 00:01:00|1556     |2972   |$-77.00|Swipe Transaction|59935      |Beulah       |ND            |58523.0|5499|NULL  |
|7475328|2010-01-01 00:02:00|561      |4575   |$14.57 |Swipe Transaction|67570      |Bettendorf   |IA            |52722.0|5311|NULL  |
|7475329|2010-01-01 00:02:00|1129     |102    |$80.00 |Swipe Transaction|27092      |Vista        |CA            |92084.0|4829|NULL  |
|7475331|2010-01-01 00:05:00|430      |2860   |$200.00|Swipe Transaction|27092      |Crown Point  |IN            |46307.0|4829|NULL  |
|7475332|2010-01-01 00:06:00|848      |3915   |$46.41 |Swipe Transaction|13051      |Harwood      |MD            |20776.0|5813|NULL  |
+-------+-------------------+---------+-------+-------+----------------+-----------+-------------+--------------+-------+----+------+
only showing top 5 rows
```

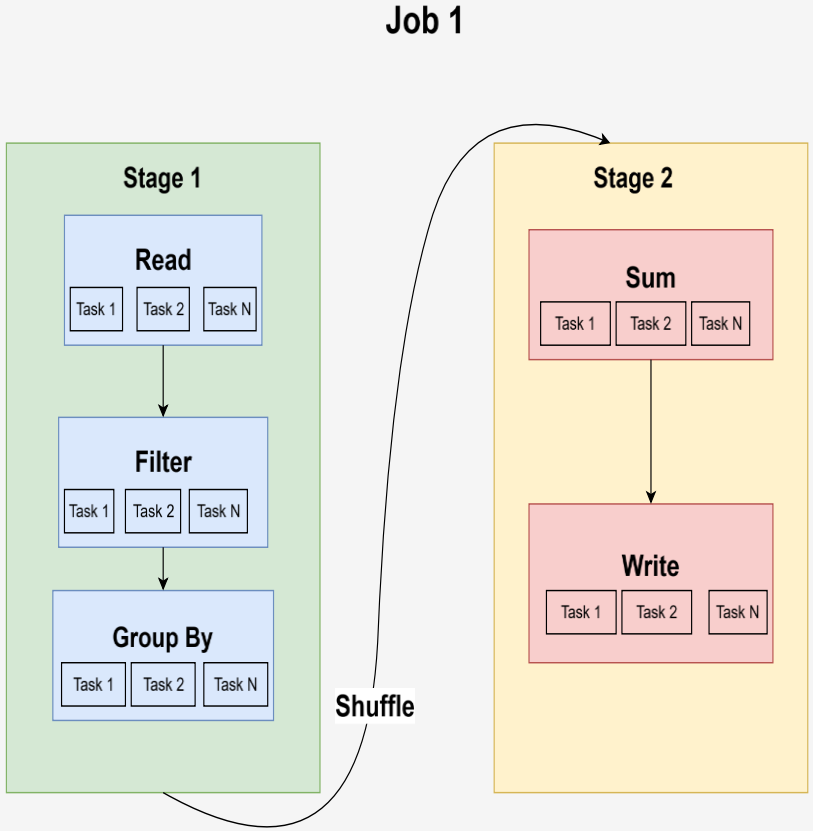
Spark Partitions

- Logical Chunk of data
- Each Partition Processed by one task(one cpu core)
- Defined degree of parallelism
- Too few partitions vs Too many partitions



Directed Acyclic Graph(DAG)

```python
# Read CSV from MinIO, file size transactions_data.csv = 1.17GB
print("Reading data from MinIO...")
df_src = spark.read \
    .option("header", "true") \
    .schema(schema) \
    .csv("s3a://dev/transactions_data.csv")

df_flt = df_src.filter("year(transaction_date) = 2014")

df_final = df_flt.groupBy('mcc').agg(
    count('id').alias('transaction_count')
)

# Write to S3
df_final.write \
    .mode('overwrite') \
    .parquet('s3a://dev/mcc_aggregated/')

print("Data is successfully loaded in s3")
```



Spark vs Pandas

Aspect	Pandas	Spark
Scale & Architecture	Single machine, in-memory (~GB range)	Distributed cluster, can handle TB/PB scale
Processing Model	Eager evaluation (immediate execution)	Lazy evaluation (builds DAG, optimizes)
Parallelism	Single-node, multi-core	Multi-node, multi-core parallelism
Data Processing	Full dataset in RAM	Partitioned across executors
Deployment	Lightweight, no cluster required	Requires Spark environment (local or cluster)
Use Cases	Ideal for analytics, ML prep, and quick exploration on local data	Ideal for ETL, big-data pipelines, and large-scale analytics

Transformation vs Action

Aspect	Transformation	Action
Execution	Lazy - doesn't execute immediately	Eager - triggers immediate execution
Purpose	Build execution plan (DAG)	Execute the DAG and return results
Returns	New DataFrame	Concrete value or output
Data Movement	Stays distributed in cluster	May bring data to driver (collect) or write to storage
Effect	Creates lineage of operations	Materializes data
Examples	select(), filter(), groupBy(), join(), withColumn()	show(), count(), collect(), write(), save()

Narrow vs Wide Transformations

Aspect	Narrow Transformation	Wide Transformation
Data Movement	Each input partition → one output partition	Each input partition → multiple output partitions
Shuffle	No shuffle	Requires shuffle across network
Performance	Fast, no network I/O	Slower, network + disk I/O overhead
Examples	map(), filter(), flatMap(), union(), coalesce()	groupBy(), join(), distinct()
Use Case	Data cleaning, projection, filtering	Aggregations, joins, sorting

Lazy Evaluation

- Spark doesn't execute transformations immediately
- Builds a logical execution plan (DAG) instead
- Actual computation happens only when an action is called
- "Plan now, execute later" strategy
- Actions – collect(), write(), show(), take()

```
# Read CSV from MinIO, file size transactions_data.csv = 1.17GB
print("Reading data from MinIO...")
df = spark.read \
    .option("header", "true") \
    .schema(schema) \
    .csv("s3a://dev/transactions_data.csv")

df_final = df.groupBy('mcc').agg(
    count('id').alias('transaction_count')
)

# Write to S3
df_final.write \
    .mode('overwrite') \
    .parquet('s3a://dev/mcc_aggregated/')

print("Data is successfully loaded in s3")
```

Dataframe and SQL Operations

```python
# Read CSV from MinIO, file size transactions_data.csv = 1.17GB
print("Reading data from MinIO...")
df = spark.read \
    .option("header", "true") \
    .schema(schema) \
    .csv("s3a://dev/transactions_data.csv")

df_final = df.groupBy('mcc').agg(
    count('id').alias('transaction_count')
)

# Write to S3
df_final.write \
    .mode('overwrite') \
    .parquet('s3a://dev/mcc_aggregated/')

print("Data is successfully loaded in s3")
```

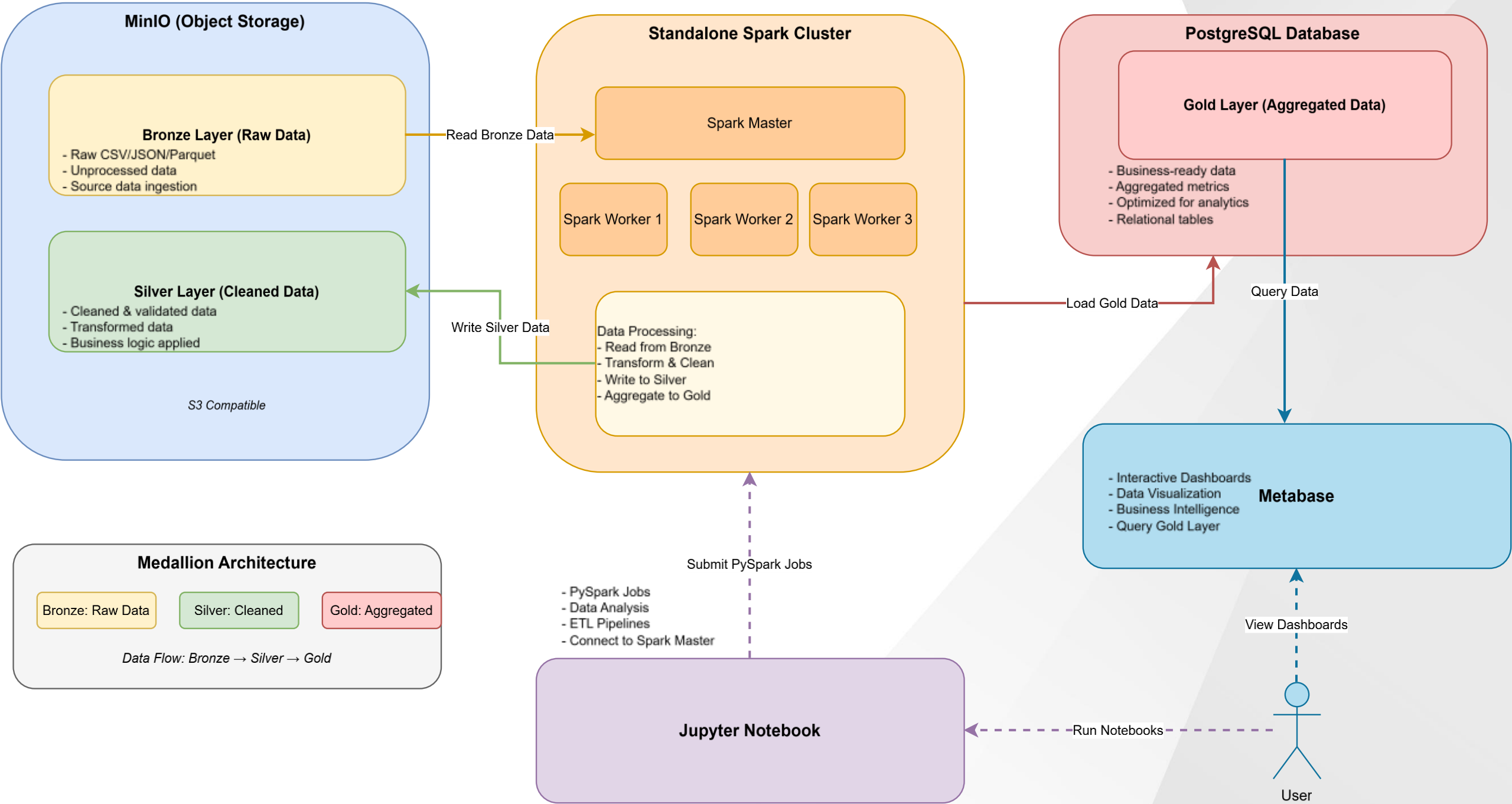
```python
# Read CSV from MinIO, file size transactions_data.csv = 1.17GB
print("Reading data from MinIO...")
df = spark.read \
    .option("header", "true") \
    .schema(schema) \
    .csv("s3a://dev/transactions_data.csv")

# Create temporary view
df.createOrReplaceTempView("transactions")

# Execute SQL query
df_final = spark.sql("""
    SELECT
        mcc,
        COUNT(id) AS transaction_count
    FROM transactions
    GROUP BY mcc
""")


# Write to S3
df_final.write \
    .mode('overwrite') \
    .parquet('s3a://dev/mcc_aggregated/')

print("Data is successfully loaded in s3")
```

When Not To Use Spark

- Small Data (<10GB)
- Low Latency Requirements (<100ms)
- OLTP Workloads
- Simple ETL on Single Machine
- Rule of Thumb:
  - **Use Spark when:** Data >10GB + Complex transformations + Batch processing
  - **Don't use Spark when:** Simple tasks + Small data + Need simplicity

Demo - Architecture

Data Pipeline Architecture - Medallion Pattern with Docker Compose



Thank You!!!

Get in touch!!!