Stony Brook University

AMS 598: Retail Inventory Management Using Big Data

Group 2

Professor: Dr. Song Wu

FAR BEYOND

Project Motivation

Motivation:

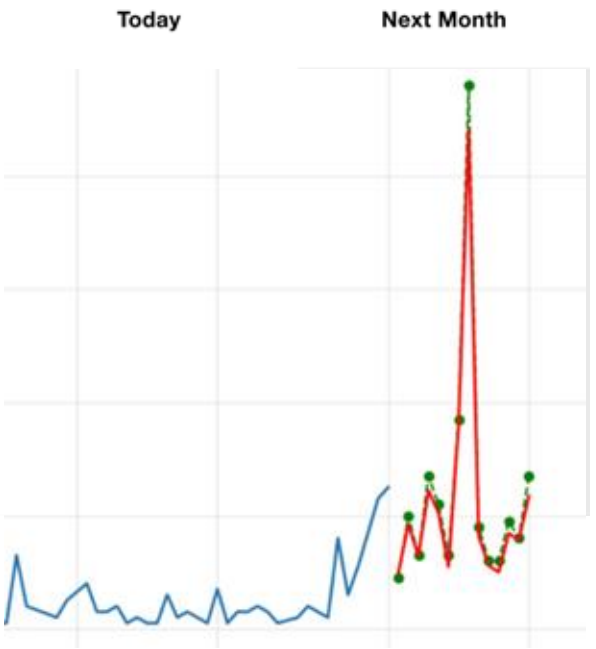
The Consumer Packaged Goods industry relies heavily on data-driven insights to optimize operations and improve decision-making. Accurate forecasting helps minimize lost sales from stockouts and reduce costs from excess inventory.

Key Decision areas:

- **Portfolio Optimization –** Selecting the right product mix across thousands of stock keeping units.
- **Inventory Management –** Ensuring products are available at the right store, in the right quantity, at the right time.

Why Big Data?

The datasets contain millions of transactions across many stores and products. Their volume, variety, and velocity require scalable methods to extract insights and support real-time forecasting.



Problem Statement

Objective:

Forecast unit_sales for the next 14 days at the store–item level, using large-scale retail sales data.

Business Problem:

- Retailers must make daily inventory decisions across thousands of stock keeping units and dozens of stores.
- Stockouts lead to lost revenue, while overstock increases holding costs and waste.
- Accurate short-term demand forecasting is essential for balancing these trade-offs.

Technical Challenge:

- The dataset contains over 23 million transactions and exhibits strong seasonality, promotions effects, and store item variability.
- Traditional single-machine methods are insufficient for processing and modeling this scale of data.

Our Goal:

Build a big-data-enabled forecasting system that learns demand patterns, predicts future unit_sales, and supports inventory optimization.

Tech Stack and Architecture

Code Framework:

- GCP, PySpark
- Random Forest forecasting model is implemented using Spark's distributed ML ecosystem.
- All datasets are stored in Parquet format.

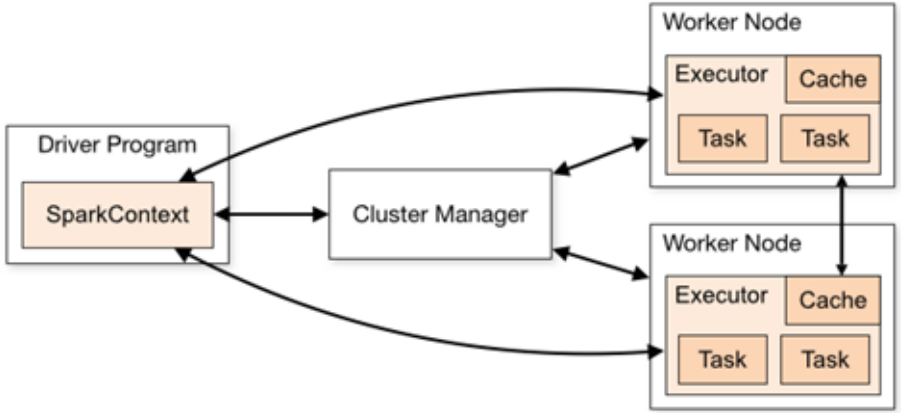
```
# Initialize Spark with Legacy Parquet Nanoseconds Config
spark = SparkSession.builder \
    .appName(CONFIG["APP_NAME"]) \
    .config("spark.sql.legacy.parquet.nanosAsLong", "true") \
    .getOrCreate()
```

Phase 1 - Data Wrangling & Preprocessing:

- Distributed Data Ingestion
- Data Cleaning and Sanitization
- Time-Series Feature Engineering
- Dataset Partitioning

Phase 2 - Model Training, Prediction & Evaluation:

- Distributed Training Pipeline
- Model Persistence & Inference
- Feature Importance Logging
- Prediction
- Performance Evaluation



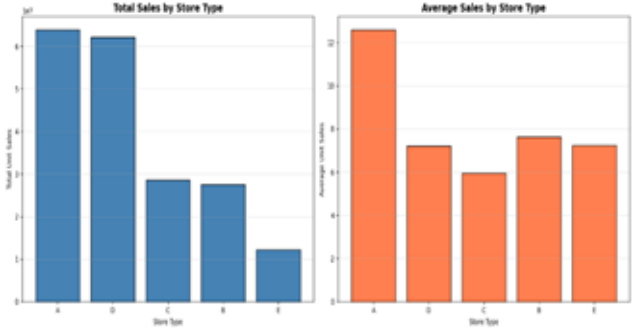
Phase 1
Data Wrangling and Preprocessing

Dataset Description

- **Source:** Corporación Favorita grocery sales dataset
- **Selected:** 2017 subset from the original dataset
- **Format:** Parquet.
- **Scale:** Approximately **23.8 million** daily store–item transaction rows.
- **Scope:**
  - Covers **54 stores** across Ecuador.
  - Includes **4,100** unique products.
  - **Unit of Data**: Each row represents the sales of a specific product in a specific store on a given day.
- **Time Range**: January 1, 2017 – August 15, 2017.

Promotion Impact on Sales

Store Type Heterogeneity



Infrastructure: Cluster Configuration - Phase 1

1. Architecture: Master-Worker Topology (YARN)

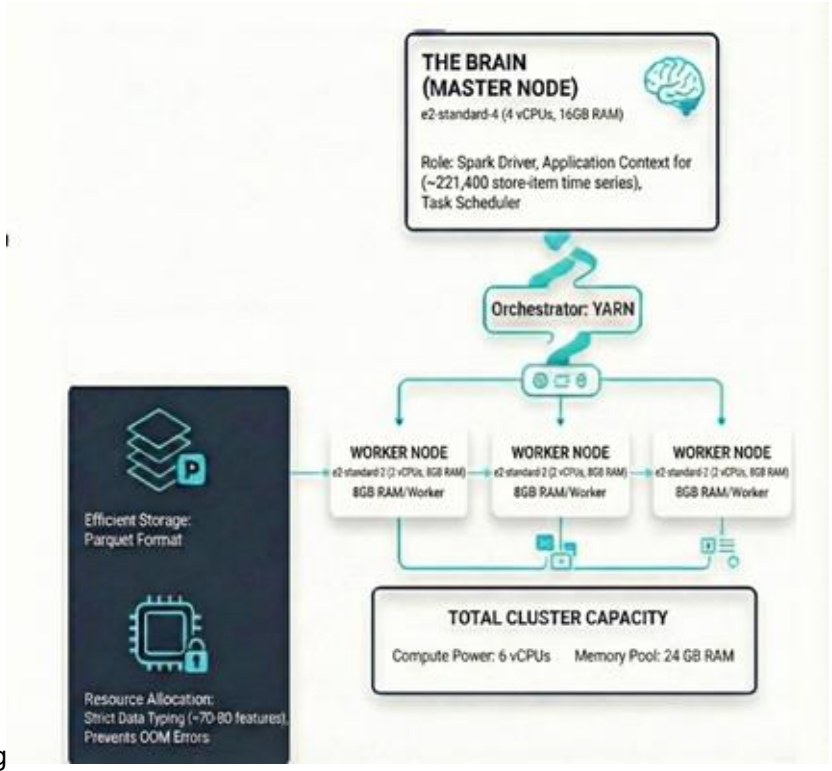
- **Orchestrator: YARN** manages resource allocation across the 3 worker nodes.
- **The Brain (Master Node):**
  - *Specs:* **e2-standard-4** (4 vCPUs, 16GB RAM).
  - *Role:* Runs the Spark Driver. It maintains the application context for the **~221,400 store-item time series** and schedules tasks (does *not* process heavy data).

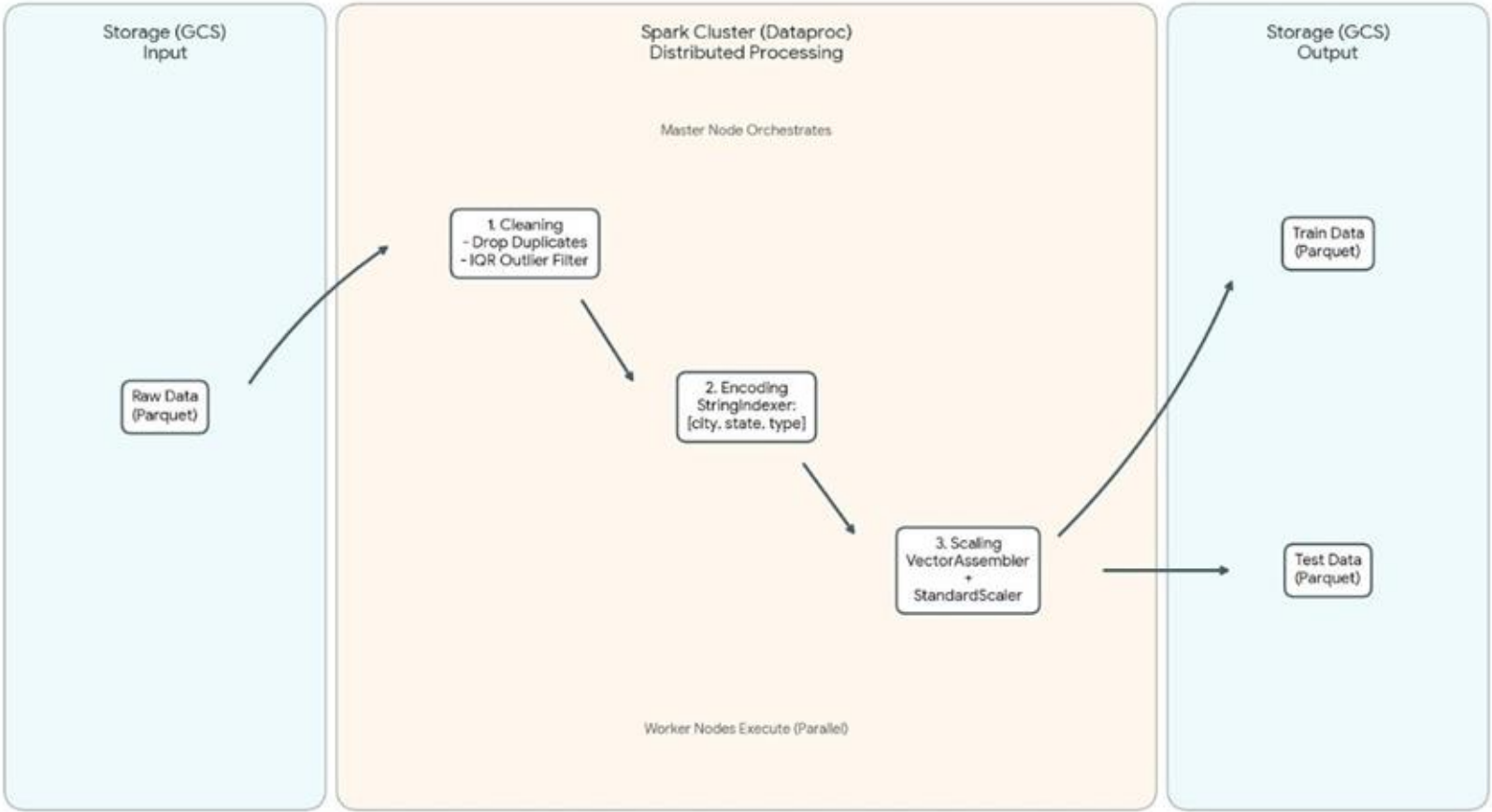
2. Total Cluster Capacity (3 Worker Nodes)

- **Compute Power:** 6 vCPUs
- **Memory Pool:** 24 GB RAM

3. Memory Optimization for Phase 1

- **Efficient Storage:** Utilization of **Parquet** format allows the smaller cluster to handle the 23.8M row dataset by minimizing I/O overhead.
- **Resource Allocation:** With 8GB RAM per worker, strict data typing was critical to prevent Out-Of-Memory (OOM) errors during the generation of **~70-80 features**.




Storage (GCS)
Input
Raw Data
(Parquet)
Spark Cluster (Dataproc)
Distributed Processing
Master Node Orchestrates
1. Cleaning
- Drop Duplicates
- IQR Outlier Filter
2. Encoding
StringIndexer:
[city, state, type]
3. Scaling
VectorAssembler
+
StandardScaler
Worker Nodes Execute (Parallel)
Storage (GCS)
Output
Train Data
(Parquet)
Test Data
(Parquet)

Data Preprocessing & Cleaning

Data Validation:

- Verified schema and data types of all fields using Spark `printSchema()`.
- Checked for missing values across all 20 columns and no missing values found.
- Checked the dataset for duplicate rows using `dropDuplicates()` and no duplicates were detected.

Data Corrections:

- Replaced negative values in numerical fields (unit_sales, unit_sales_raw, transactions) with 0, since negative values are not meaningful in retail sales.

Outlier Analysis:

- Performed IQR-based outlier detection on all numerical fields using `approxQuantile()`
- Identified high-outlier columns such as: unit_sales, unit_sales_raw (≈ 9.7% outliers), transactions, is_holiday, class, log_sales
- Outliers were not removed because large spikes correspond to real retail demand events (e.g., holidays, promotions, payday cycles).

Categorical Processing: Used StringIndexer to encode: city, state, type, family.

Numerical Scaling:

- Created standardized features (Z-score).
- Created min–max normalized features (0–1).

Feature Engineering

Feature Categories:

Category	Features	Description
Lag Features (7)	t-1, t-2, t-3, t-7, t-14, t-21, t-28	Historical sales at various time lags
Rolling Statistics (12)	Mean, std, min, max for 7/14/30-day windows	Aggregated metrics excluding current row to prevent leakage
EWMA (3)	Spans: 7, 14, 30 days	Exponentially weighted moving averages for trend detection
Change Features (4)	Day-over-day, week-over-week (absolute & %)	Sales differences and percentage changes
Temporal (11)	Year, month, day, DOW, DOY, WOY, quarter, is_weekend, is_month_start/end	Calendar-based features and cyclical patterns
Store-Item Patterns (5)	DOW/month averages, promo lags (7/14 days), 30-day promo frequency	Store-item specific behavioral patterns
Trend (2)	days_since_first_sale, sales_momentum_7	Long-term trends and momentum indicators

Train-Test Split:Train: First 213 days | **Test:** Last 14 days (forecasting horizon) | **Target:** unit_sales

Phase 2
Model Training, Prediction & Evaluation

Infrastructure: Cluster Configuration - Phase 2

1. Architecture: Master-Worker Topology (YARN)

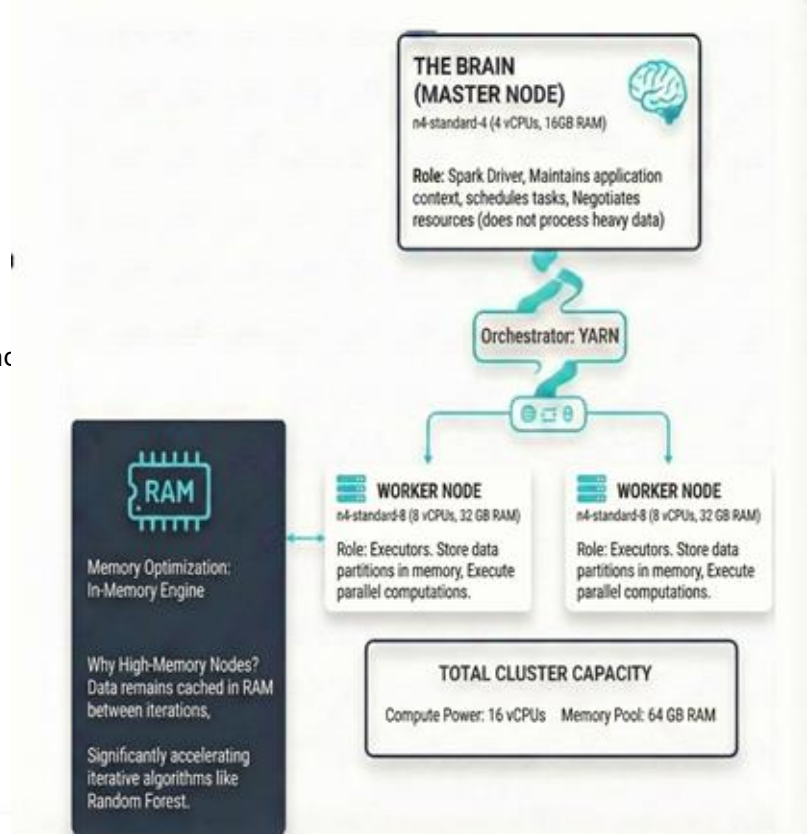
- **Orchestrator: YARN** manages resource allocation.
- **The Brain (Master Node):**
  - *Specs:* **n4-standard-4** (4 vCPUs, 16GB RAM).
  - *Role:* Runs the Spark Driver. Maintains application context, schedules tasks, and negotiates resources
- **The Muscle (2x Worker Nodes):**
  - *Specs:* **n4-standard-8** (8 vCPUs, 32 GB RAM per node).
  - *Role:* **Executors**. These nodes store data partitions in memory and execute parallel computations.

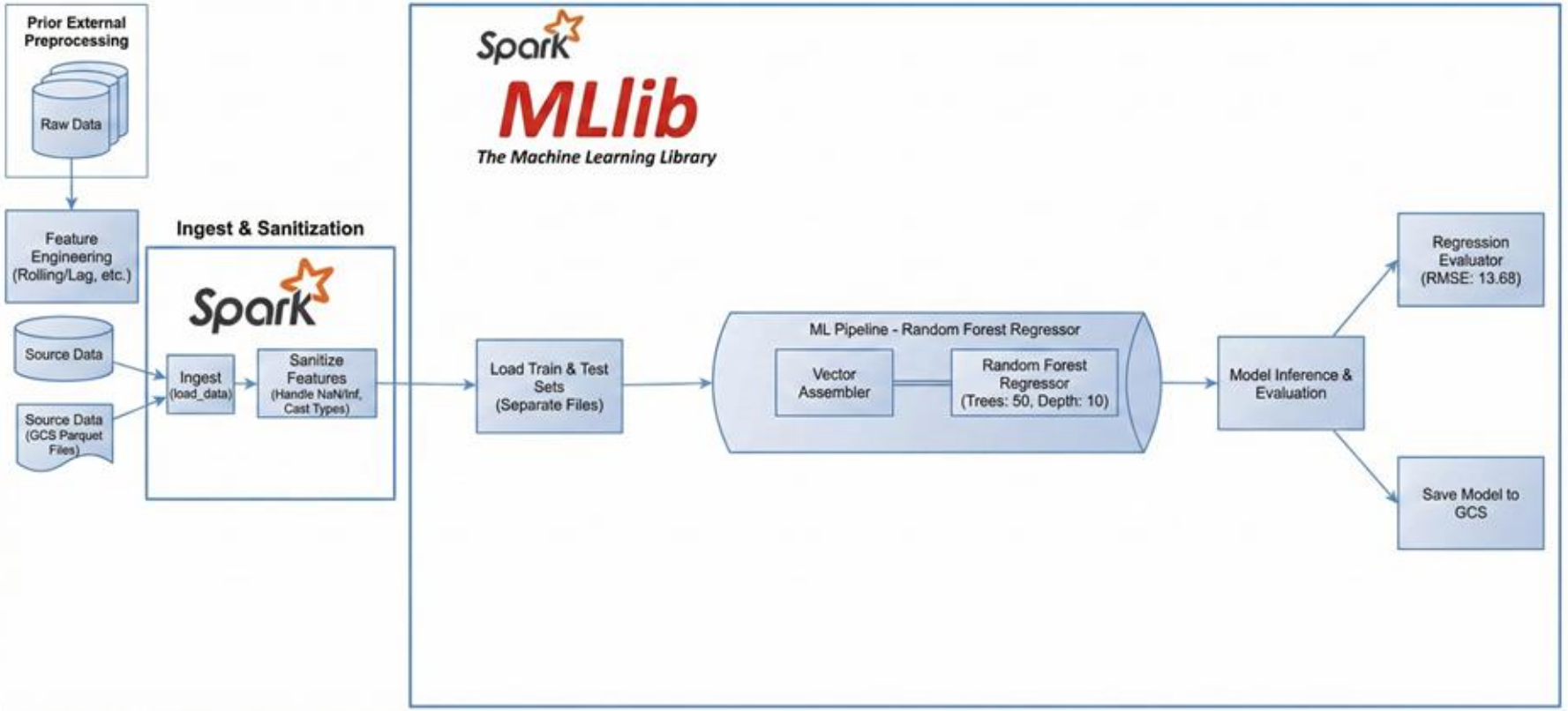
2. Total Cluster Capacity

- **Compute Power:** 16 vCPUs
- **Memory Pool:** 64 GB RAM

3. Memory Optimization

- **Why High-Memory Nodes?** Unlike Hadoop MapReduce (disk-based), Spark is an **In-Memory** engine.
- **Benefit:** Data remains cached in RAM between iterations, significantly accelerating iterative algorithms like **Random Forest**.




Prior External Preprocessing
Raw Data
Feature Engineering (Rolling/Lag, etc.)
Source Data
Source Data (GCS Parquet Files)
Ingest & Sanitization
Spark
Ingest (load_data)
Sanitize Features (Handle NaN/Inf, Cast Types)
Spark
MLlib
The Machine Learning Library
Load Train & Test Sets (Separate Files)
ML Pipeline - Random Forest Regressor
Vector Assembler
Random Forest Regressor (Trees: 50, Depth: 10)
Model Inference & Evaluation
Regression Evaluator (RMSE: 13.68)
Save Model to GCS

Data Pipeline: Storage & Optimization

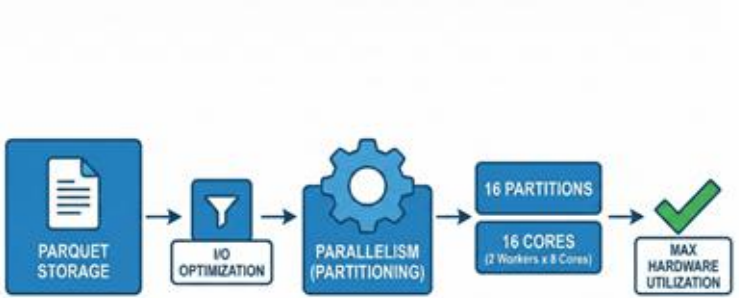
1. Storage Format: Parquet

- **Why Parquet:** Superior compression and strict schema preservation compared to CSV.
- **I/O Optimization:** Leverages Pushdown:
  - **Projection Pushdown:** Spark reads only required columns.
  - **Predicate Pushdown:** Filters are applied at the storage layer; irrelevant file blocks are skipped entirely.

2. Parallelism Strategy: Partitioning

- **Configuration:** `df.repartition(16)`
- **The Logic:** Cluster Capacity = 2 Workers × 8 Cores = **16 Total Cores**.
- **Optimization Goal:** Achieved a **1:1 Task-to-Core Ratio**.
  - Too Few (<16): Idle CPUs
  - Too Many (>16): High scheduler overhead

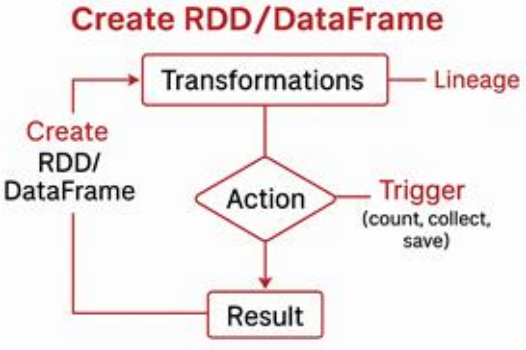
Result: Maximum hardware utilization



Spark Execution & Model Preparation

The Paradigm

- **Concept:** Transformations are not executed immediately. Spark records them as a Logical Plan.
- **Trigger:** Execution only occurs when an Action is called.
- **Benefit of Lazy Evaluation:** This delay allows Spark to optimize the plan before execution by reordering operations to minimize data movement and processing cost.



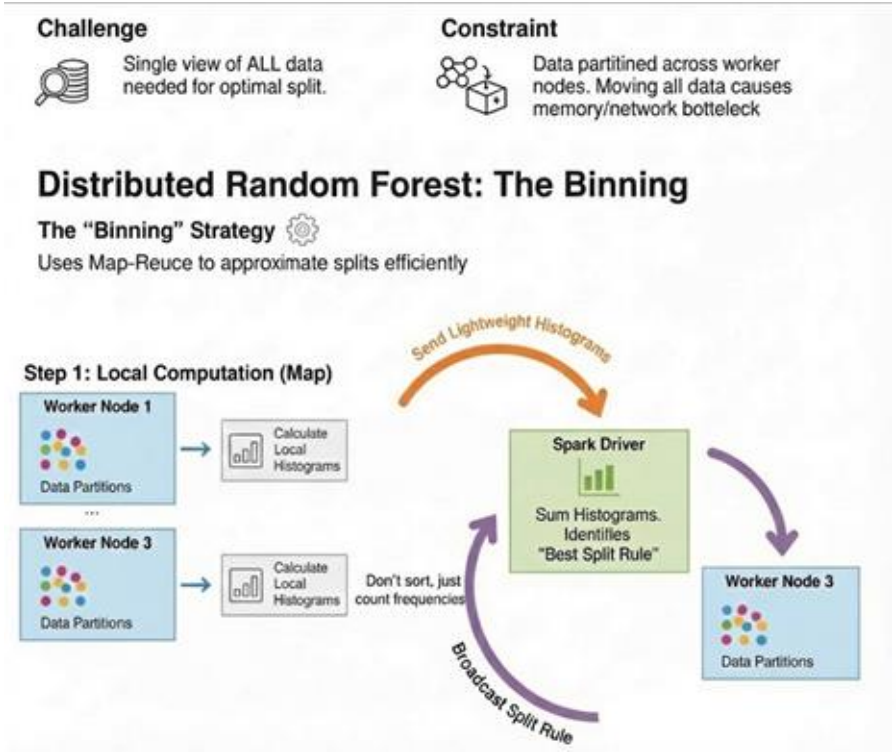
Data Formatting

- **The Requirement:** Unlike Scikit-Learn which accepts Matrices or DataFrames, Spark MLlib requires features to be condensed into a Single Vector Column.
- **Implementation:** Used `VectorAssembler` to fuse individual feature columns (Store ID, Item ID, Sales Lags) into a compact feature vector for the model.

```
# 3. Build Pipeline
assembler = VectorAssembler(
    inputCols=feature_cols,
    outputCol="features",
    handleInvalid="keep"
)
```

Transformation: `[Col A, Col B, Col C]` → `[Vector(A, B, C)]`

Distributed Random Forest



Problem:

Finding the **globally optimal split rule** for decision trees is **bottlenecked** by the need to **aggregate all data partitioned across worker nodes.**

Solution:

- **Map:**

Worker nodes compute and send lightweight local histograms (compressed data summaries) instead of moving all data.

- **Reduce/Driver:**

The Spark Driver aggregates these histograms to find the best approximate split rule quickly.

- **Synchronization:**

The new split rule is broadcast back to all workers to continue tree growth.

Result:

Efficiently approximates the optimal split with minimal network communication.

Distributed Model Training

1. Stage 109 (Map): Parallel Computation

- **Data Parallelism:** Workers read partitions directly from disk.
- **Local Bootstrapping:** Independent sampling with replacement.

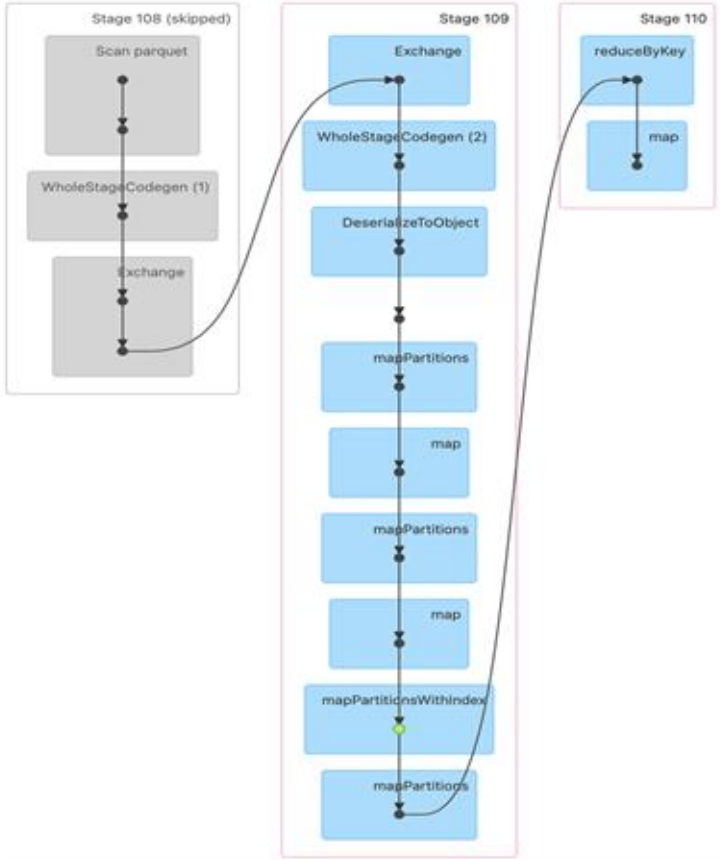
2. The Shuffle (Exchange)

- **Network Barrier:** Partial metadata transmitted to synchronize nodes.
- **Optimization:** Only summary statistics shuffled (no raw data).

3. Stage 110 (Reduce): Aggregation

- **Global Aggregation:** `reduceByKey` sums partial histograms.
- **Driver Sync:** `collectAsMap` returns the "best split" to the Driver to update the tree.

Code moves to data, not the other way around. *Heavy computation runs locally where data resides. Only lightweight summary statistics are shuffled, minimizing network latency.*



This DAG visualizes how Spark leverages the MapReduce to build decision trees across multiple worker nodes simultaneously.

Code Snippet - Main Logic (RF) - Training & Prediction

```python
# 3. Build Pipeline
assembler = VectorAssembler(
    inputCols=feature_cols,
    outputCol="features",
    handleInvalid="keep"
)

rf = RandomForestRegressor(
    featuresCol="features",
    labelCol="unit_sales",
    numTrees=CONFIG["NUM_TREES"],
    maxDepth=CONFIG["MAX_DEPTH"],
    seed=CONFIG["SEED"]
)

pipeline = Pipeline(stages=[assembler, rf])

# 4. Train
print("Starting Random Forest training...")
model = pipeline.fit(train_df)
print("Training completed.")
```

```python
# 2. Clean Test Data (Must apply same cleaning as training!)
test_df, _ = sanitize_features(test_df, EXCLUDED_COLS)

# 3. Load Model
print(f"Loading model from {CONFIG['MODEL_OUTPUT_PATH']}...")
loaded_model = PipelineModel.load(CONFIG["MODEL_OUTPUT_PATH"])
print("Model loaded.")

# 4. Generate Predictions
print("Running inference...")
predictions = loaded_model.transform(test_df)

# 5. Calculate RMSE
evaluator = RegressionEvaluator(
    labelCol="unit_sales",
    predictionCol="prediction",
    metricName="rmse"
)
rmse = evaluator.evaluate(predictions)
print(f"Root Mean Squared Error (RMSE): {rmse:.4f}")
```

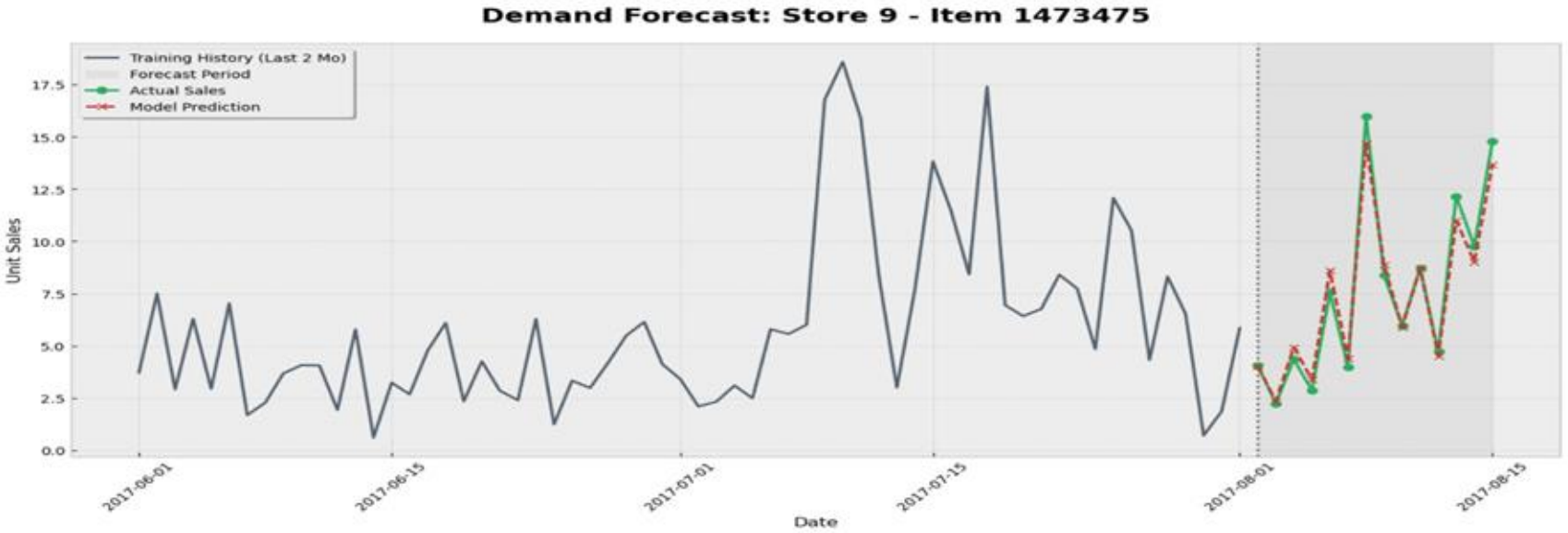
Results & Discussion

Model Performance

- **Metric: RMSE ≈ 13.69.**
- **Interpretation:** On average, predictions deviate by ~13 inventory units. This variance indicates the baseline for calculating minimum extra Stock requirements to prevent stockouts

Feature Importance

- **Primary Driver:** The Lagged Variables - `store_item_month_avg_sales` .
- **Model Insight:** The model identified historical sales behavior as the strongest predictor of future demand.



THANK YOU