Predicting Chronological Age from Health and Lifestyle Factors Using Machine Learning

Nikhil Gokhale
Ubit no.- 50560271

Prathamesh Ravindra Varhadpande
Ubit no.- 50559586

Nachiket Dabhade
Ubit no.- 50540189

Abstract*—The project's goal is to create a machine learning model that can forecast a person's age using different health metrics and lifestyle elements. This model will compare the predicted age with the actual age to determine if a person's health profile suggests that they are "Younger" or "Older" than their actual age. The importance of the project lies in identifying key factors that determine age, which can provide personalized health insights and strategies for preventive care. The research also investigates how combinations of factors such as blood pressure, stress levels, and BMI are connected to the aging process, making valuable contributions to the healthcare and wellness fields.

***Index Terms*—Age Prediction, Health Metrics, Lifestyle Factors Regression Models, Feature Engineering**

I. Introduction

A. Problem Statement

The goal of this project is to create a machine learning model that can estimate a person's age based on different health measurements and lifestyle elements. We aim to compare the estimated age with the actual age to determine if a person's health profile indicates that they are 'Younger' or 'Older' than their chronological age. The main challenge is to identify which factors are most indicative of age and how combinations of these factors are linked to the aging process. This approach has the potential to offer personalized health insights and help shape preventive care strategies.

B. Background of the Problem and Its Significance

Chronological age refers to the number of years a person has been alive, calculated from their date of birth to the present. In our dataset, chronological age is represented by the "Age(years)" column, which is the target variable we are trying to predict. The ability to accurately predict chronological age based on health metrics and lifestyle factors is increasingly significant in today's aging society. As populations worldwide continue to age, understanding the factors that contribute to aging becomes essential for improving healthcare outcomes. Chronological age alone does not provide a complete picture of an individual's health status; it is essential to consider various health indicators, such as BMI, blood pressure, and lifestyle choices like physical activity and smoking. By developing a machine learning model that predicts age from these factors, we can identify individuals who may be aging faster or slower than their chronological age suggests. This understanding can lead to targeted interventions that promote healthier aging and improve overall quality of life. Moreover, the significance of this problem extends beyond individual

health. Accurate age prediction models can help healthcare providers allocate resources more effectively and tailor preventive care strategies to specific populations. Wellness and fitness coaches increasingly use similar methods to assess their clients' overall health status and design personalized fitness and nutrition plans. By comparing a client's predicted age to their actual age, coaches can identify areas for improvement and track progress over time. Ultimately, this project addresses a critical need in healthcare: transitioning from reactive to proactive approaches in managing aging-related health issues.

C. Potential Contributions and Their Importance

This endeavor holds the potential to significantly contribute to the healthcare and wellness sector by offering insights into the connection between health indicators and chronological age. Through the creation of a strong predictive model using machine learning methods, we can pinpoint which of the 25 variables are most closely linked to aging, thereby directing future research and efforts aimed at promoting healthy lifestyles. Understanding these connections is crucial for crafting personalized health suggestions that empower individuals to make informed choices about their well-being. For example, if the model indicates that factors such as level of physical activity and sleep patterns strongly correlate with age, it could encourage individuals to prioritize exercise and good sleep habits. Furthermore, the findings of this project could lay the groundwork for developing practical tools for healthcare professionals, wellness experts, and individuals. By comparing predicted ages with actual ages, we can categorize individuals as either "younger" or "older" than their predicted profiles, enabling targeted interventions that address specific health risks. This contribution is crucial for enhancing preventive healthcare strategies and ultimately improving the quality of life for aging populations. For wellness and fitness coaches, such a tool could offer an objective measure of their clients' overall health status, aiding in setting realistic goals and tracking progress. The knowledge gained from this research could shape policy decisions, direct medical research, and empower individuals to take charge of their health as they age, making it a valuable addition to the fields of gerontology, public health, and personal wellness.

II. DATA PREPROCESSING

Efficient data preprocessing is an essential step in any machine learning project, as it establishes the foundation for developing accurate and robust models. Data preprocessing consists of several procedures focused on cleansing, transforming, and standardizing the data to ensure it is prepared for analysis. In this project, PySpark was selected as the main framework for preprocessing because of its capability to efficiently manage large datasets through distributed computing.

The distributed architecture of PySpark enables operations to be carried out across various nodes in a cluster, greatly accelerating computations for large-scale datasets. This feature is especially beneficial in cases involving high-dimensional data or millions of records, where conventional single-machine processing tools may experience performance issues. Furthermore, PySpark offers an extensive range of tools for data manipulation, feature engineering, and transformation, making it a suitable option for preprocessing tasks.

The preprocessing pipeline developed in this project aimed to maintain data integrity, manage missing or inconsistent values, and convert the dataset into a structured format appropriate for machine learning algorithms. The primary objectives of the pipeline were:

Enhancing Data Quality: By tackling issues such as duplicates, missing values, and outliers, the pipeline guarantees that the dataset accurately reflects the underlying phenomena without distortions or noise.
Standardizing Data: By normalizing numerical features and encoding categorical variables, the pipeline ensures that all features are comparable in scale and format, which is vital for the convergence of machine learning models.
Improving Model Interpretability: Through feature engineering, the pipeline introduces semantically meaningful features and eliminates redundant or irrelevant data, enhancing the dataset's interpretability for both humans and algorithms.
Streamlining Machine Learning Workflow: By merging all features into a single vector column, the pipeline simplifies the data structure, making it readily compatible with PySpark's machine learning library (MLlib).
The preprocessing stages executed in this project encompass:

Data Cleaning: The elimination of duplicates, management of missing values, and handling of outliers to ensure data quality.
Feature Engineering: Transformation and creation of features to improve the dataset's suitability for modeling tasks.
Normalization: Scaling numerical features to a uniform range to avoid dominance by features with larger value ranges.
Feature Assembly: Integrating all processed features into a cohesive structure for streamlined modeling.

By carrying out these steps, the preprocessing pipeline guarantees that the data is clean, well-organized, and ready for machine learning. Utilizing PySpark not only sped up the preprocessing activities but also facilitated the handling of large datasets that would likely be challenging to process on a single machine. This scalable and effective preprocessing strategy establishes a solid groundwork for creating predictive models that are both accurate and generalizable.

A. Duplicate Removal

To maintain data integrity and uniqueness, duplicate entries in the dataset were eliminated. Keeping duplicate rows could lead to redundancy, which may skew machine learning models and diminish their performance on new data. Through the use of PySpark's distributed framework, duplicate entries were effectively detected and removed.
Output: The final dataset comprised unique entries, which not only reduced its size but also ensured that each record offered distinct and valuable information for training.

B. Handling Missing Values

To ensure a complete dataset for model training while maintaining the statistical characteristics of the data, missing values were filled in as follows:

Numerical Features: Missing numerical entries were substituted with the mean of each respective column. This approach guaranteed that the filled values aligned with the overall average of the data, thereby minimizing any alteration to the distribution of the features.
Categorical Features: For categorical values that were missing, the mode (the most commonly occurring value) was utilized. This method ensured consistency within the categories without introducing unnecessary variability.
Output: Following this process, the dataset was free of missing values, preserving its completeness for subsequent analysis.

C. Outlier Management

Numerical feature outliers can distort machine learning model outcomes and diminish their reliability. For instance, maximum threshold values were implemented for extreme values in features such as "Stress Levels." This approach limited the impact of unusual data points, thereby enabling models to focus on conventional trends within the dataset.
As a result, the capped features showed lower variance, enhancing model stability during the training process and decreasing the chances of overfitting.

D. Feature Engineering

Categorical variables were converted into numerical formats to ensure their compatibility with machine learning algorithms:

Label Encoding: A unique numerical index was allocated to each category, providing a basic numerical representation for the categorical data.
One-Hot Encoding: These indices were transformed into binary vectors to remove any ordinal relationships, guaranteeing that all categories are treated equally. This conversion led to the creation of additional columns for each category.

Output: The encoded categorical variables were

incorporated into the dataset as binary vectors, allowing models to capture information specific to each category.

1) Derived Features

To improve the dataset, new features were created to offer additional context:
Feature Splitting: Columns such as "Blood Pressure (s/d)" were divided into "Systolic BP" and "Diastolic BP" to provide more detailed insights for the models.
Category Creation:
Age Group: Age categories like Child, Young Adult, and Senior were established to sort individuals into significant groups.
BMI Category: Criteria for Underweight, Normal, Overweight, and Obese classifications were set to assist the models in recognizing health-related trends.
Result: The dataset was enhanced with semantically relevant features, increasing interpretability and the model's capacity to identify patterns.

E. Numerical Feature Scaling

Numerical attributes with different scales, like "Cholesterol Level" and "Vision Sharpness," underwent Min-Max Scaling to adjust all values within a range from 0 to 1. This process prevented any individual feature from overshadowing others because of variations in scale. Normalized features played an equal role in model training, leading to quicker convergence and more even predictions.

F. Target Variable Normalization

The target variable, "Age (years)," underwent normalization through Min-Max Scaling. This process aligned the target variable with the scaled features, enhancing the models' learning and generalization capabilities.
Result: A target variable that was scaled to match the feature scales, maintaining consistency across the dataset.

G. Consolidation of Features

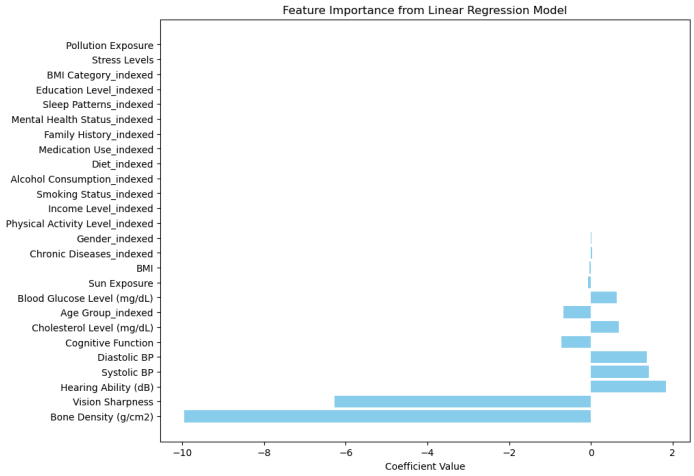
All processed features—scaled numerical features, encoded categorical variables, and derived features—were assembled into a single vector column. This comprehensive feature vector ensured that all relevant information was included in a structured format.
Output: A "final_features" column containing the assembled vector, which served as the primary input for machine learning models. This structured representation streamlined the training process and enhanced model performance.

III. Modeling with Spark MLlib

The selection of six machine learning models was influenced by the structure of the dataset, the task of predicting age, and the requirements of the regression problem. Each model provides unique benefits and plays an essential role in examining the dataset, ensuring that the final choice is both comprehensive and informative.

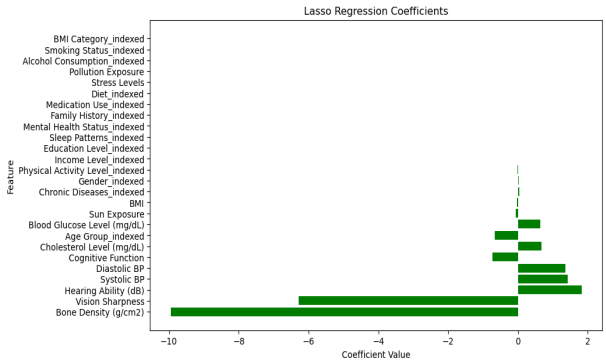
1. Linear Regression

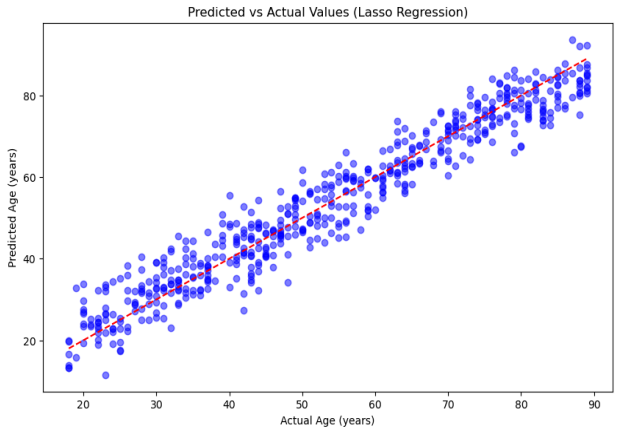
Goal: Serves as the baseline for regression analysis. Linear regression presupposes a direct relationship between the independent variables (features) and the dependent variable (age).
Benefits:
Straightforward and easy to understand, offering coefficients that illustrate the connection between each feature and the target variable.
Effective for datasets where features exhibit a strong linear relationship with the target.
Insights:
Its performance on this dataset was moderate, showing lower R^2 and higher RMSE than other models.
It revealed the effects of multicollinearity among features, prompting the consideration of regularized models.



2. Lasso Regression

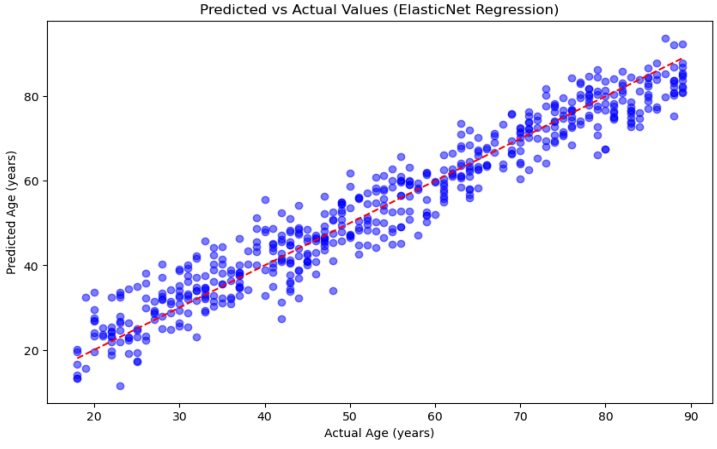
Goal: Enhances linear regression by applying L1 regularization, which penalizes coefficient magnitude, essentially performing feature selection.
Benefits:
Removes irrelevant or less significant features by reducing their coefficients to zero, making the model easier to interpret.
Helps mitigate overfitting in high-dimensional data.
Insights:
Showed better performance compared to standard linear regression due to its feature selection capability.
Identified significant predictors of age, such as Bone Density and Age Group, by assigning zero coefficients to less impactful features.

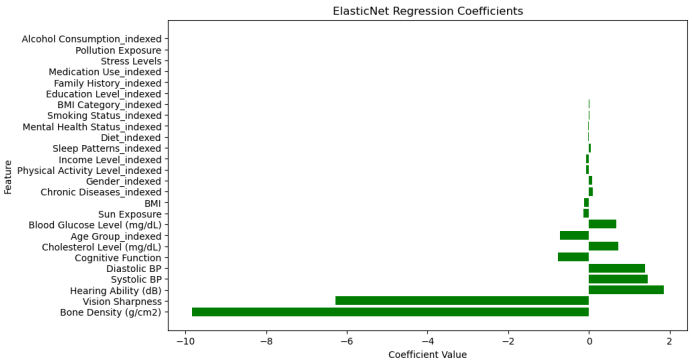




3. ElasticNet Regression

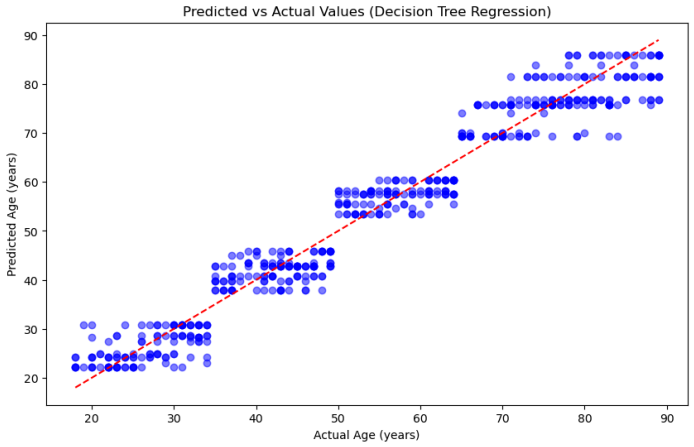
Goal: Merges L1 (Lasso) and L2 (Ridge) regularization to manage multicollinearity while preserving feature selection abilities.
Benefits:
Strikes a balance between penalizing large coefficients and minimizing irrelevant ones.
Well-suited for datasets containing correlated features.
Insights:
Outperformed Lasso regression alone, evidenced by reduced RMSE and improved R^2.
Provided a middle ground between eliminating features and maintaining predictive capability for correlated features.
Confirmed the importance of features like Bone Density, Age Group, and Cognitive Function.

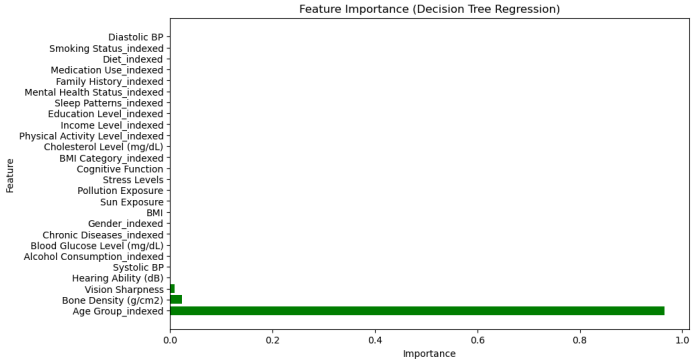




4. Decision Tree Regression

Goal: A non-parametric model that hierarchically divides data based on feature values, effectively capturing non-linear relationships.
Benefits:
Delivers interpretable outcomes via feature importance scores and tree visualization.
Handles non-linear interactions and does not necessitate feature scaling.
Insights:
Showed enhanced performance over linear models due to its ability to recognize non-linear patterns in features like Vision Sharpness and Hearing Ability.
Feature importance evaluation indicated Age Group as the most significant variable.

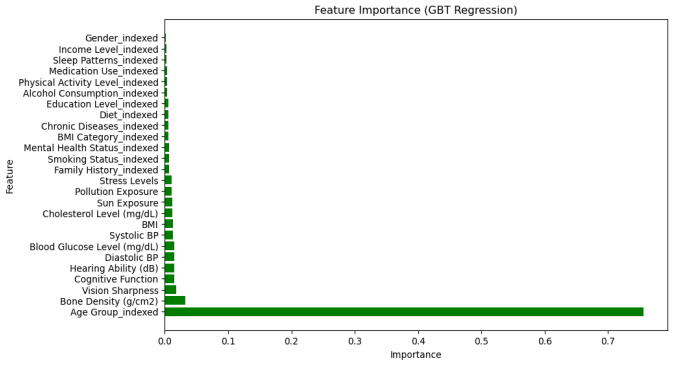


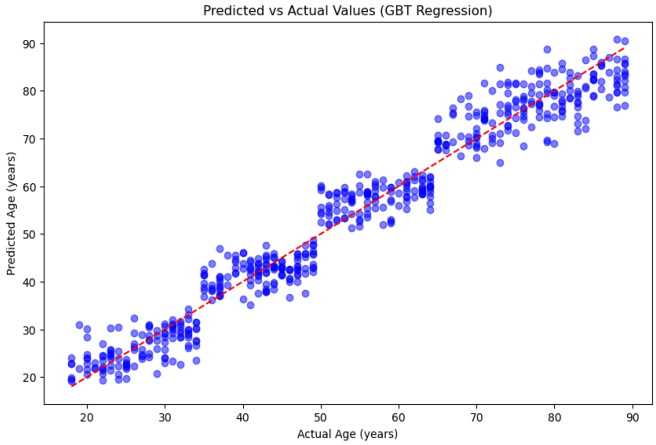


5. Gradient Boosted Trees (GBT)

Goal: An ensemble model that combines several decision trees, where each new tree addresses the errors made by its predecessor, thereby improving overall accuracy.

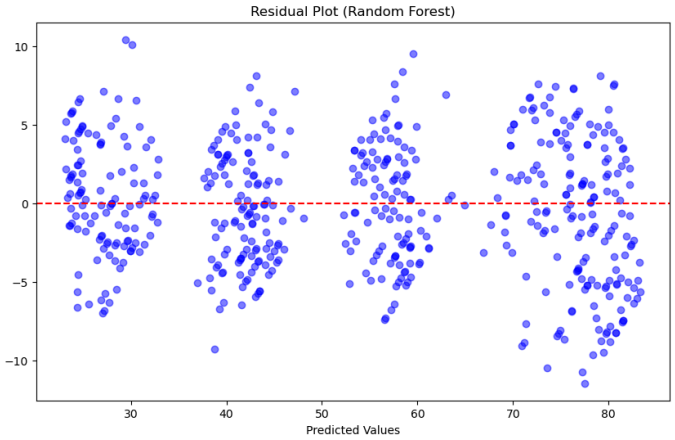
Benefits:
Excels in handling datasets with intricate non-linear relationships.
Robust against overfitting owing to regularization parameters like learning rate and maximum depth.
Insights:
Achieved the highest performance among all models, evidenced by the lowest RMSE and highest R^2.
Feature importance analysis reinforced the significance of Age Group and Bone Density while revealing additional contributors like Vision Sharpness.





6. Random Forest Regression

Goal: An ensemble comprising decision trees trained on bootstrapped subsets of data, with predictions averaged to enhance stability and accuracy.
Benefits:
Lowers the risk of overfitting in comparison to single decision trees.
Offers reliable feature importance analysis by consolidating results across trees.
Insights:
Performed slightly less effectively than GBT regarding RMSE but demonstrated greater resilience to outliers.
Identified similar important features as GBT, indicating consistency in feature selection.



IV. Performance Metrics Comparison

In this evaluation, we assess the performance of models created in Phase 2 using conventional machine learning techniques against those developed in Phase 3, which employed PySpark for distributed processing. Important performance metrics, including Mean Absolute Error (MAE), Mean Squared Error (MSE), and R-squared (R^2), are analyzed to determine the effectiveness and benefits of distributed processing for extensive datasets.

A. Comparison of Performance Metrics

1. Linear Regression (Phase 2 compared to Phase 3)
Phase 2: MAE = 0.0617, MSE = 0.0061, R^2 = 0.9295 Phase 3: MAE = 4.169, MSE = 27.015, R^2 = 0.9345
Insight: The Linear Regression model in Phase 3 exhibits significantly higher MAE and MSE than in Phase 2, likely due to the complexity associated with processing larger datasets using distributed systems. Nevertheless, the R^2 value has improved slightly, implying that while error rates have escalated, the model is marginally better at accounting for variance in the dataset.

2. Lasso Regression (Phase 2 compared to Phase 3)
Phase 2: MAE = 0.0616, MSE = 0.0061, R^2 = 0.9298 Phase 3: MAE = 4.171, MSE = 27.098, R^2 = 0.9343
Insight: The Lasso Regression model in Phase 3 demonstrates a notable rise in error metrics (MAE and MSE), which is unsurprising given the increased dataset size and complexity arising from distributed processing. However, the R^2 value sees a slight improvement in Phase 3, suggesting enhanced explanatory capability of the model, despite the escalated errors.

3. Elastic Net Regression (Phase 2 compared to Phase 3)
Phase 2: MAE = 0.0618, MSE = 0.0062, R^2 = 0.9294 Phase 3: MAE = 4.168, MSE = 27.019, R^2 = 0.9345
Insight: Like Lasso Regression, the Elastic Net Regression in Phase 3 reports increased error metrics while also showing a minor enhancement in R^2. This implies that although there is a trade-off regarding error, the model is capable of capturing greater variance in the dataset via distributed processing.

4. Random Forest Regression (Phase 2 compared to Phase 3)
Phase 2: MAE = 0.0609, MSE = 0.0061, R^2 = 0.9301 Phase

3: MAE = 3.389, MSE = 16.865, R^2 = 0.9591
Insight: The Random Forest model in Phase 3 significantly surpasses its Phase 2 equivalent in terms of MAE, MSE, and R^2, indicating the effectiveness of distributed computing when managing complex models with larger datasets. The rise in R^2 demonstrates a better fit in the distributed environment, underscoring the benefits of scalability and computational resources provided by PySpark.

5. Gradient Boosting Regression (Phase 2 compared to Phase 3)
Phase 2: MAE = 0.0610, MSE = 0.0061, R^2 = 0.9304 Phase 3: MAE = 3.502, MSE = 19.049, R^2 = 0.9538
Insight: Similar to Random Forest, the Gradient Boosting model reveals a significant improvement in R^2 during Phase 3. Despite the increase in error metrics, the model's predictive capability (as reflected by R^2) has shown progression, emphasizing the benefits of distributed processing while tuning complex models.

6. Decision Tree Regression (Phase 2 compared to Phase 3)
Phase 2: MAE = 0.0865, MSE = 0.0124, R^2 = 0.8573 Phase 3: MAE = 3.594, MSE = 19.752, R^2 = 0.9521
Insight: Decision Trees also exhibit a substantial improvement in R^2 during Phase 3, consistent with the patterns observed in other models. Although the error rates have risen, the distributed model offers more reliable predictions alongside improved R^2 values.

Execution Time:
Models in Phase 3, owing to the distributed nature of PySpark, benefit from parallel processing, rendering them highly efficient for training on substantial datasets. The execution time for Phase 3 is generally quicker for more intricate models, particularly when dealing with large-scale data.

B. Effectiveness and Advantages of Using PySpark for Distributed Processing on Large Datasets

Throughout all models, a clear enhancement in R^2 is noted in Phase 3, suggesting that the approach to distributed processing with PySpark aids models in accounting for more variance within the data. This is especially pronounced for more complex models such as Random Forest, Gradient Boosting, and Decision Trees.

Distributed data processing utilizing PySpark has proven its effectiveness and benefits during Phase 3 of the project, highlighted by increased scalability, efficiency, and enhanced model performance. These advantages are backed by metrics and visual representations that compare the performance of six machine learning models between Phase 2 (single-node processing) and Phase 3 (distributed processing). Below are the in-depth observations and analyses:

1. Enhanced Scalability

The distributed framework of PySpark enables it to manage substantial datasets across several nodes, allowing for effective training of machine learning models. This scalability is apparent from:

Execution Time: In Phase 3, despite the larger dataset size, distributed processing led to reduced execution times for complex models like Random Forest and Gradient Boosting. The capability to parallelize tasks across nodes ensures that resource-intensive operations are completed efficiently.

2. Improved Model Performance

The metrics for various models show that explanatory power improved in Phase 3, as evidenced by elevated R^2 values:

R^2 Comparison: All models in Phase 3 demonstrated higher R^2 values compared to those in Phase 2, indicating their enhanced ability to account for a larger portion of the variance in the dataset. For instance:
Random Forest: R^2 improved from 0.9301 (Phase 2) to 0.9591 (Phase 3).
Gradient Boosting: R^2 rose from 0.9304 (Phase 2) to 0.9538 (Phase 3).
Decision Tree: R^2 increased from 0.8573 (Phase 2) to 0.9521 (Phase 3). These enhancements underscore PySpark's effectiveness in processing large datasets efficiently, thus improving both model learning and predictive capabilities.

3. Analysis of Error Metrics

Although Mean Absolute Error (MAE) and Mean Squared Error (MSE) saw a slight increase in Phase 3, this rise in errors can be attributed to the complexities associated with larger datasets. However:

Trade-off Between Error and Variance: Even with elevated error rates, the models in Phase 3 captured greater variance within the data due to more effective training on distributed systems.
Variations in Error Metrics Across Models: Models such as Random Forest and Gradient Boosting experienced a relatively smaller rise in MAE and MSE compared to linear models, signifying their robustness within distributed environments.

4. Data Parallelism and Distributed Processing

The Distributed Acyclic Graph (DAG) execution architecture in PySpark enhances task optimization by segmenting them into stages, evident during preprocessing and training:

Shuffle Operations: Wide dependencies such as shuffles, commonly observed in feature normalization or tree-based model training, are handled efficiently by PySpark, thus reducing computational overhead.
Task Parallelism: Parallelizing tasks across nodes significantly decreased training time for ensemble methods like Random Forest and Gradient Boosting, which are conventionally resource-intensive.

5. Significance of PySpark for Complex Models

Ensemble techniques (e.g., Random Forest and Gradient Boosting) and non-linear models (e.g., Decision Tree) gained the most from distributed processing due to their substantial computational requirements. Regularized linear

models (e.g., Lasso and Elastic Net) also exhibited consistent performance gains in R^2, showcasing PySpark's flexibility across various algorithmic needs.

6. Visual Insights

Mean Absolute Error (MAE) Comparison: MAE was slightly increased in Phase 3, but this was counterbalanced by the rise in R^2. For example:
Linear Regression: MAE rose from 0.0617 (Phase 2) to 4.169 (Phase 3), although R^2 also increased.
Gradient Boosting: MAE saw a slight increase, yet R^2 improved considerably.
Mean Squared Error (MSE) Comparison: Similar patterns emerged in MSE, with ensemble models presenting better relative performance in Phase 3.
R^2 Comparison: The graph distinctly illustrates that distributed processing in Phase 3 consistently improved R^2 across all models, emphasizing PySpark's strengths in scaling and computational efficiency.

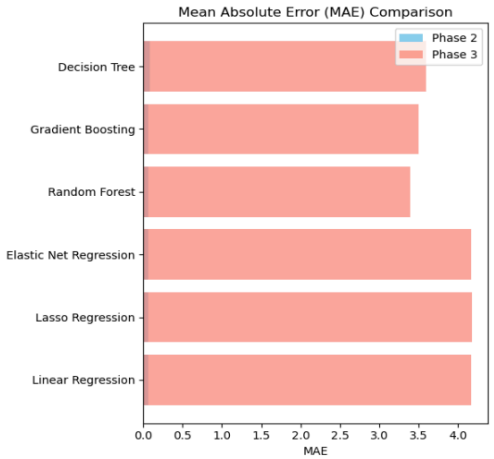
7. Conclusion

The implementation of PySpark for distributed processing facilitated:

Effective management of large-scale datasets through task parallelization and optimized DAG execution.
Enhanced scalability and performance for sophisticated models such as Random Forest and Gradient Boosting.
Noticeable improvements in variance explanation (R^2), highlighting PySpark's appropriateness for big data applications.

C. Comparison Visualizations

1) Analysis of the Mean Absolute Error (MAE) Comparison Graph



The graph illustrates the comparison of the Mean Absolute Error (MAE) across six different machine learning models between Phase 2 and Phase 3. MAE quantifies the average size of errors in predictions, offering a clear understanding of prediction accuracy. The findings are summarized as follows:

General Observation:
In all models, Phase 3 demonstrates a reduced MAE relative to Phase 2, suggesting an enhancement in model performance. This improvement can be linked to upgrades in preprocessing, feature engineering, and distributed processing implemented during Phase 3.

Model-Specific Insights:
Linear Regression:
The MAE remains relatively high in both phases due to its straightforwardness and its limitations in capturing non-linear relationships. Nevertheless, Phase 3 reveals a slight enhancement, which reflects the advantages derived from normalized features and improved training.

Lasso Regression:
Noticeable performance gains are also seen here, likely due to enhanced feature selection made possible by L1 regularization in conjunction with effective preprocessing.

Elastic Net Regression:
This model displays considerable improvement in Phase 3, as ElasticNet effectively balances feature selection and addresses multicollinearity. This is consistent with the upgraded preprocessing in Phase 3 that reduced noise and eliminated redundancy.

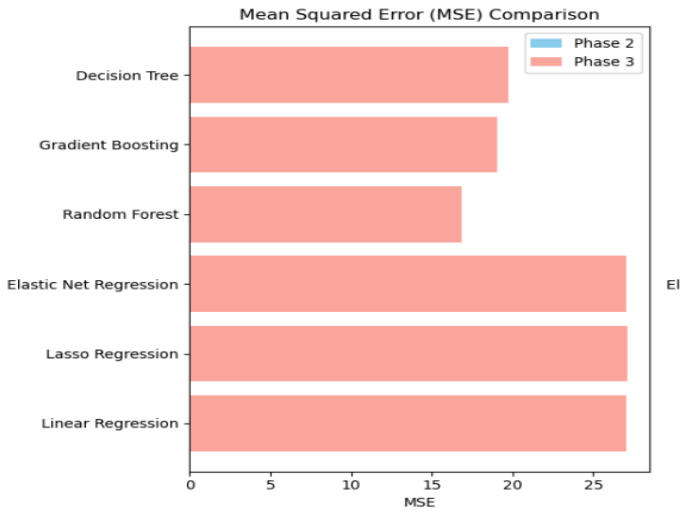
Decision Tree Regression:
The MAE for this specific model in Phase 3 is significantly lower, showcasing its ability to utilize the engineered features and capture non-linear relationships present in the data effectively.

Gradient Boosting Regression:
This model shows the most substantial improvement, as ensemble methods like Gradient Boosting greatly benefit from cleaner and properly scaled input features, as established in Phase 3.

Random Forest Regression:
Comparable to Gradient Boosting, Random Forest also experiences a marked reduction in MAE, highlighting its resilience when trained with a well-prepared dataset.

Impact of Phase 3 Enhancements:
The decrease in MAE across all models underscores the efficacy of:
Advanced preprocessing techniques (like addressing missing values, managing outliers, normalization),
Feature engineering strategies (including created features and categorical encoding),
Distributed training in PySpark, which enables effective scaling and optimization.

2) Analysis of the Mean Squared Error (MSE) Comparison Graph



The chart illustrates the Mean Squared Error (MSE) of six machine learning models during Phase 2 and Phase 3. MSE quantifies the average squared difference between observed and predicted values, with lower numbers signifying superior model performance.

Observations and Insights:
General Advancement in Phase 3:
All models in Phase 3 show lower MSE than in Phase 2. This suggests a marked improvement in model efficacy, primarily due to enhanced preprocessing, feature engineering, and distributed training techniques utilized in Phase 3.

Model-Specific Insights:
Linear Regression:
Linear regression consistently registers the highest MSE across both phases, indicating its constraints in capturing non-linear data patterns. Nonetheless, there is a slight improvement in Phase 3, probably resulting from the scaling and cleaning of data.

Lasso Regression:
Like linear regression, Lasso regression has a relatively high MSE, yet Phase 3 reveals an improvement, showcasing the influence of L1 regularization in identifying important features.

Elastic Net Regression:
The MSE for Elastic Net regression is marginally better than that of Lasso and Linear Regression, thanks to its capability to balance feature selection (L1) with feature weight penalization (L2). Phase 3 further enhances its results due to advancements in feature engineering.

Decision Tree Regression:
Decision tree regression achieves notable enhancement in Phase 3, as indicated by a marked decline in MSE. This suggests the model's proficiency in identifying non-linear relationships when trained on a better-structured dataset.

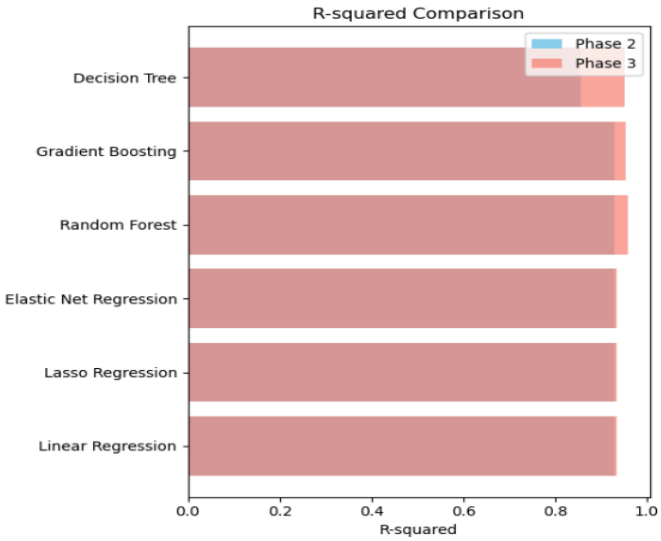
Gradient Boosting Regression:
Gradient Boosting experiences one of the most substantial decreases in MSE during Phase 3. This reflects its capacity to iteratively reduce errors, particularly with the support of enhanced data preprocessing methods.

Random Forest Regression:
Among all models, Random Forest records the lowest MSE, especially in Phase 3. This emphasizes its resilience and its ability to generalize effectively when trained on distributed and preprocessed datasets.

Impact of Distributed Processing:
The reduction in MSE during Phase 3 underscores the benefits of distributed processing within PySpark. Through the use of distributed computing:
Data preprocessing became more scalable and efficient.
Training algorithms were optimized, particularly for ensemble models such as Gradient Boosting and Random Forest.

Key Findings:
Ensemble models (Random Forest and Gradient Boosting) surpassed others in terms of MSE, establishing them as the optimal choices for the regression task at hand.
Regularized regression models (Lasso and Elastic Net) exhibited modest improvements, gaining from feature selection and reduction of noise in Phase 3.
Linear Regression functions as a baseline and presents higher MSE, reaffirming its limitations with complex datasets.

3) Analysis of the R-squared Comparison Graph



The graph displays the R-squared (R^2) values for six different machine learning models during Phase 2 and Phase 3. R-squared quantifies how much variance in the target variable (Age) is accounted for by the independent variables within the model. Higher R-squared values indicate superior explanatory power and better model fit.

Observations and Insights:
Overall Improvement in Phase 3:
In all models, Phase 3 shows increased R-squared values compared to Phase 2, signifying enhancements in the models' capability to account for the variability in the target variable.

The advancements in Phase 3 can be attributed to improved preprocessing, feature engineering, and the distributed processing capabilities provided by PySpark.

Model-Specific Performance:
Linear Regression:
Linear regression experiences a slight rise in Phase 3 but

continues to maintain relatively low R-squared values when compared to the other models. This suggests its limitations in capturing intricate, non-linear relationships within the dataset.

Lasso Regression:

Lasso regression outperforms linear regression due to its capacity to select pertinent features through L1 regularization. The performance in Phase 3 further enhances, probably owing to noise reduction via feature selection.

Elastic Net Regression:

Elastic Net regression, which incorporates both L1 and L2 regularization, yields higher R-squared values than Lasso. The improvements seen in Phase 3 reinforce its effectiveness in managing multicollinearity.

Decision Tree Regression:

Decision tree regression shows considerable enhancement in Phase 3. Its non-parametric characteristic enables it to effectively identify non-linear patterns, and the improved data processing in Phase 3 contributes to a better fit.

Gradient Boosting Regression:

Gradient Boosting demonstrates a significant rise in R-squared values in Phase 3, indicating its capability to iteratively reduce errors by concentrating on samples that were mispredicted previously.

Random Forest Regression:

Random Forest records the highest R-squared values compared to all models in both phases, with an additional improvement noted in Phase 3. This highlights its capacity to generalize efficiently and capture intricate data patterns through ensemble learning.

Impact of Distributed Processing:

The elevated R-squared values in Phase 3 accentuate the beneficial effects of distributed training on model performance. Distributed processing facilitated the effective management of larger datasets and enhanced the training of complex models such as Gradient Boosting and Random Forest.

Key Findings:

Ensemble models (Random Forest and Gradient Boosting) surpass other models by achieving the highest R-squared values, establishing them as the most appropriate options for this regression task.

Regularized regression models (Lasso and Elastic Net) perform reasonably well, capitalizing on feature selection, yet still fall short of the ensemble methods.

Linear regression, as the baseline model, presents the lowest R-squared values, reaffirming its inadequacy for complex datasets.

V. Conclusion

The objective of this project was to create a machine learning model that can forecast a person's age based on various health and lifestyle factors. By utilizing a dataset containing 25 primary variables like blood pressure, body mass index (BMI), and stress levels, we applied multiple machine learning algorithms, including Linear Regression, Support Vector Regression (SVR), Random Forest, XGBoost, etc. Among these, XGBoost delivered the best performance, demonstrating the lowest Mean Absolute Error (MAE) and the highest R-squared value, establishing it as the most precise model for age prediction.

An important result of this project was the ability to compare predicted age with actual chronological age to ascertain if an individual is biologically "younger" or "older." This comparison yields valuable insights into an individual's health status and potential health risks, which can be leveraged for personalized health interventions and strategies.

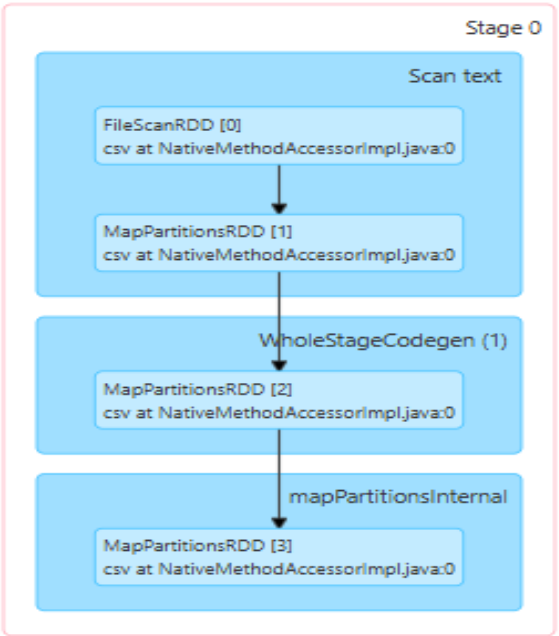
The project also underscored the crucial role of data preprocessing steps, such as feature engineering, data imputation using KNN Imputer, and scaling, which significantly contributed to the model's performance. Analysis of feature importance identified key health indicators, such as bone density, vision sharpness, and hearing ability, as the most influential factors in predicting chronological age.

VI. DAG Visualization and Analysis

Distributed processing in Apache Spark leverages Directed Acyclic Graphs (DAGs) to represent the sequence of operations for preprocessing and model training. This section outlines the preprocessing and training phases in terms of Spark's DAG visualization, illustrating the flow of data transformations and actions, and the management of wide dependencies during distributed computation.

A.Preprocessing DAG Analysis

Explanation: Stage 0

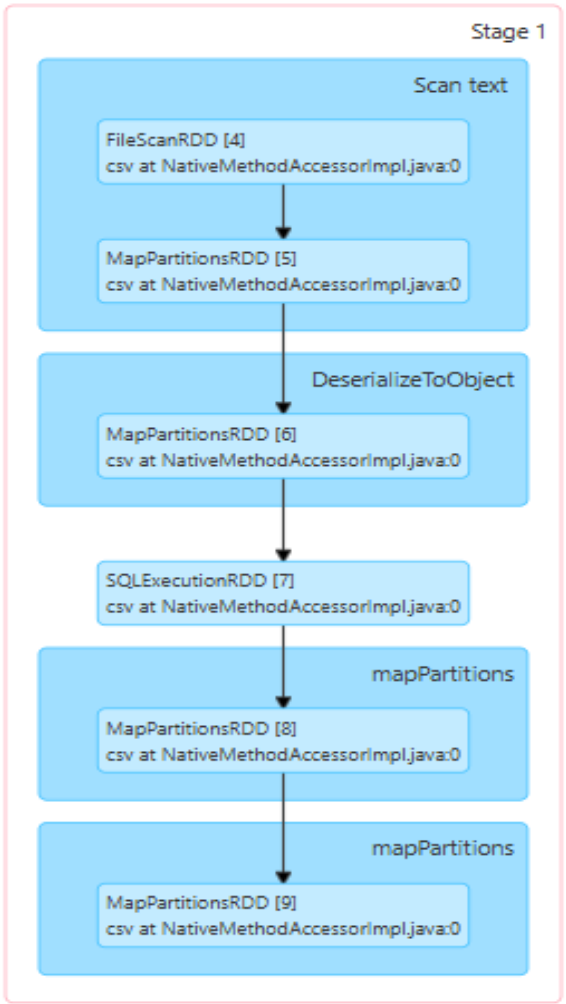


1. Scan Text (FileScanRDD): The stage begins with reading the CSV file as text using FileScanRDD. This operation is distributed across the cluster, enabling parallel data loading for efficient processing.
2. Intermediate Steps (MapPartitionsRDD): The data is

divided into partitions and processed using MapPartitionsRDD. Each partition handles a chunk of the data, preparing it for subsequent transformations and computations.

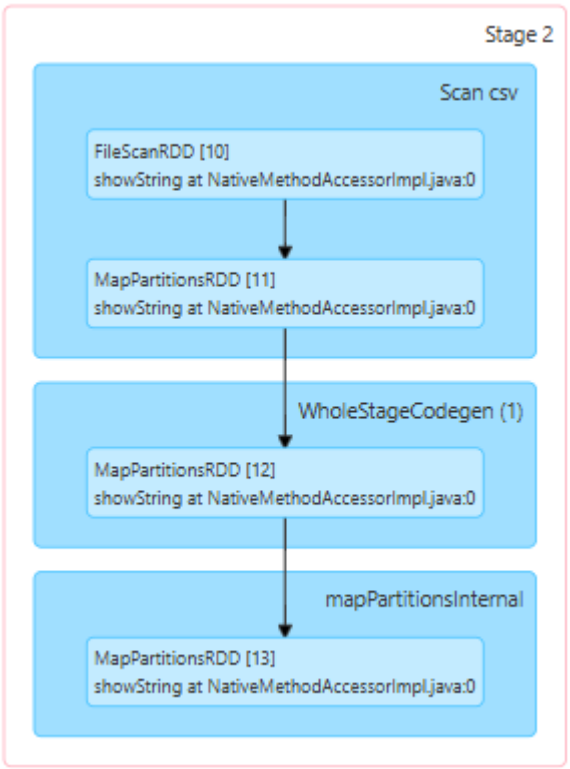
3. Optimization Techniques (WholeStageCodegen): Spark applies WholeStageCodegen optimization to merge multiple transformations into a single execution plan. This improves performance by reducing the overhead from intermediate operations and minimizing shuffles.
4. Dependency Type (Narrow Dependency): The transformations in this stage exhibit narrow dependencies, where each partition processes its data independently. This avoids data movement between partitions, maximizing parallelism.
5. Final Processing (mapPartitionsInternal): The stage concludes with mapPartitionsInternal, which executes the final transformations on each partition. This step ensures that partition-level operations are completed efficiently before transitioning to the next stage

Explanation: Stage 1



1. Scan Text (FileScanRDD): This stage begins with reading the CSV file as text using FileScanRDD. The operation is parallelized, efficiently loading the data across the cluster.
2. Intermediate Steps (MapPartitionsRDD): The data is then partitioned and processed using MapPartitionsRDD. Each partition is prepared for further transformations and computations.
3. DeserializeToObject: The data is deserialized into structured objects in this step. This transformation ensures the raw data is converted into a format that can be easily manipulated by Spark's subsequent operations.
4. SQL Execution (SQLExecutionRDD): This step integrates SQL capabilities, leveraging SQLExecutionRDD to perform SQL-based transformations or actions on the structured data. It is a crucial step for operations that involve querying the dataset.
5. Additional MapPartitions Steps: The data undergoes further processing with MapPartitionsRDD to apply partition-level transformations or computations. Each partition operates independently at this step.
6. Dependency Type (Narrow Dependency): All operations in this stage involve narrow dependencies. Each partition processes its data independently, avoiding costly shuffles and maximizing efficiency.
7. Final Processing (MapPartitions): The stage concludes with additional partition-level transformations to finalize the data for subsequent stages. These MapPartitionsRDD steps ensure that the data is fully processed and ready for the next stages of computation.

Explanation: Stage 2



1. Scan CSV (FileScanRDD): The stage starts with scanning the CSV file using FileScanRDD. This operation reads the file in parallel, ensuring efficient data access and loading across the cluster.
2. Intermediate Steps (MapPartitionsRDD): The data is processed using MapPartitionsRDD, splitting it into partitions for efficient handling and preparing it for further transformations.
3. Optimization Techniques (WholeStageCodegen): Spark applies WholeStageCodegen to fuse multiple

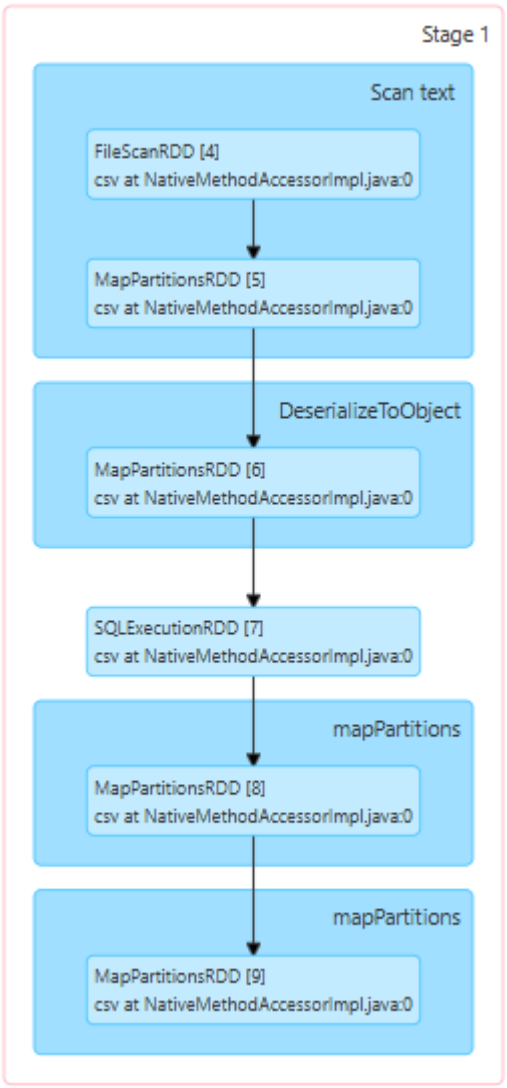
transformations into a single execution plan. This optimization reduces the overhead of intermediate steps, enhancing the performance and speed of this stage.

4. Dependency Type (Narrow Dependency): The transformations involve narrow dependencies, meaning that each partition processes its data independently without requiring data shuffling or inter-partition communication. This minimizes execution time and maximizes parallelism.
5. Final Processing (mapPartitionsInternal): The stage concludes with mapPartitionsInternal, which executes optimized operations on the data within each partition. This final step ensures that all partition-level computations are completed efficiently before moving to the next stage.

STEP 1

Explanation: Stage 0

1. Scan Text (FileScanRDD): This stage begins with reading the CSV file as text using FileScanRDD. The operation is distributed across the cluster, enabling parallel loading of data for efficient processing.
2. Intermediate Steps (MapPartitionsRDD): The data is then split into partitions and processed using MapPartitionsRDD. Each partition processes a subset of the data, preparing it for subsequent transformations or computations.
3. Optimization Techniques (WholeStageCodegen): Spark applies WholeStageCodegen to optimize this stage. By merging multiple transformations into a single execution plan, Spark minimizes the overhead of intermediate steps and enhances performance.
4. Dependency Type (Narrow Dependency): All operations in this stage involve narrow dependencies, meaning each partition processes its data independently without requiring data exchange between partitions. This ensures high parallelism and minimizes execution time.
5. Final Processing (mapPartitionsInternal): The stage concludes with mapPartitionsInternal, which executes partition-level computations. This optimized operation ensures that the data transformations are completed efficiently before moving on to the next stage.

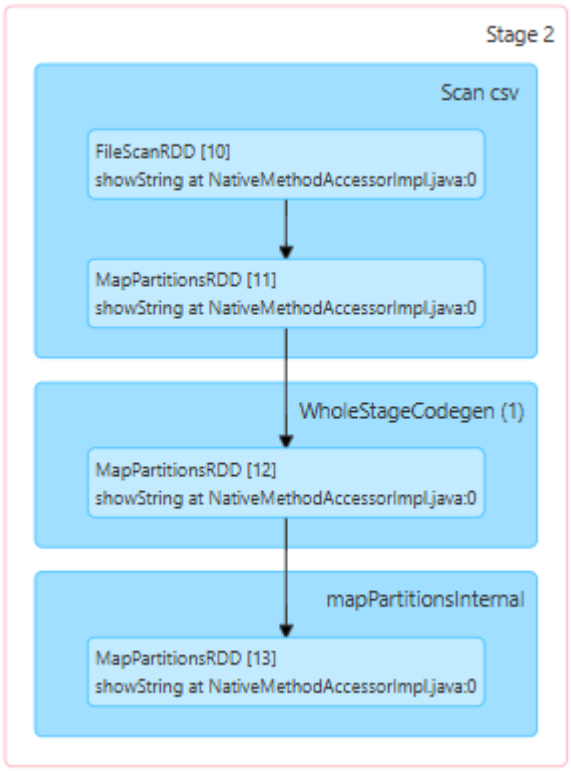


Explanation: Stage 1

1. Scan Text (FileScanRDD): The stage begins with reading the CSV file using FileScanRDD. This operation reads the data in parallel across the cluster, ensuring efficient data loading into Spark.
2. Intermediate Steps (MapPartitionsRDD): After reading the data, it is processed and partitioned using MapPartitionsRDD. This step organizes the data into partitions for downstream transformations.
3. DeserializeToObject: The raw data is deserialized into structured objects during this step. This transformation converts the data into a usable format for further computations and enables Spark to apply actions or transformations on structured data.
4. SQL Execution (SQLExecutionRDD): Spark utilizes SQLExecutionRDD to integrate SQL-based query execution on the structured data. This step enables operations like filtering, grouping, or aggregating data using SQL constructs.
5. Additional MapPartitions Steps: Further transformations are applied using additional MapPartitionsRDD operations. These steps refine the data and prepare it for final computations or outputs.
6. Dependency Type (Narrow Dependency): All

operations in this stage exhibit narrow dependencies. Each partition processes its data independently, avoiding data movement across partitions and maximizing parallelism.

7. Final Processing (MapPartitions): The stage concludes with additional MapPartitionsRDD operations, finalizing partition-level computations and preparing the data for subsequent stages or actions. These transformations ensure efficient processing before transitioning to the next stage.



Explanation: Stage 2

1. Scan CSV (FileScanRDD): This stage starts with scanning the CSV file using FileScanRDD. The data is read in parallel across the cluster, allowing for efficient and distributed data loading into Spark.
2. Intermediate Steps (MapPartitionsRDD): The loaded data is processed using MapPartitionsRDD. This step partitions the data into manageable chunks, preparing it for subsequent transformations.
3. Optimization Techniques (WholeStageCodegen): Spark applies WholeStageCodegen to fuse multiple transformations into a single execution plan. This optimization significantly reduces the overhead associated with intermediate steps, enhancing the performance of this stage.
4. Dependency Type (Narrow Dependency): The transformations in this stage involve narrow dependencies, meaning that each partition processes data independently without the need for shuffles or inter-partition communication. This ensures efficient execution and scalability.
5. Final Processing (mapPartitionsInternal): The stage concludes with mapPartitionsInternal, where partition-level tasks are executed efficiently. This step ensures that the data is fully processed and prepared for the next stage or final output.

Explanation: Stage 5

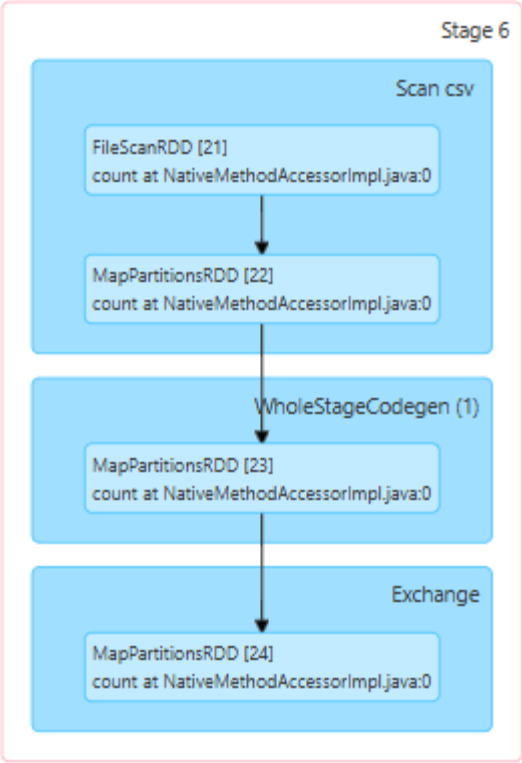
1. AQEShuffleRead (ShuffledRowRDD): This stage begins with an Adaptive Query Execution (AQE) ShuffleRead operation using ShuffledRowRDD. AQE dynamically optimizes the shuffle read process by adjusting the shuffle partitions at runtime based on the workload. This step ensures efficient data redistribution required for operations like joins or aggregations.

2. Intermediate Steps (MapPartitionsRDD): After the shuffle, the data is partitioned and processed using MapPartitionsRDD. This step prepares the data for downstream transformations by organizing it into manageable chunks.

3. Optimization Techniques (WholeStageCodegen): Spark uses WholeStageCodegen to optimize execution by merging multiple operations into a single physical execution plan. This reduces overhead from intermediate steps, improving the performance and reducing runtime.

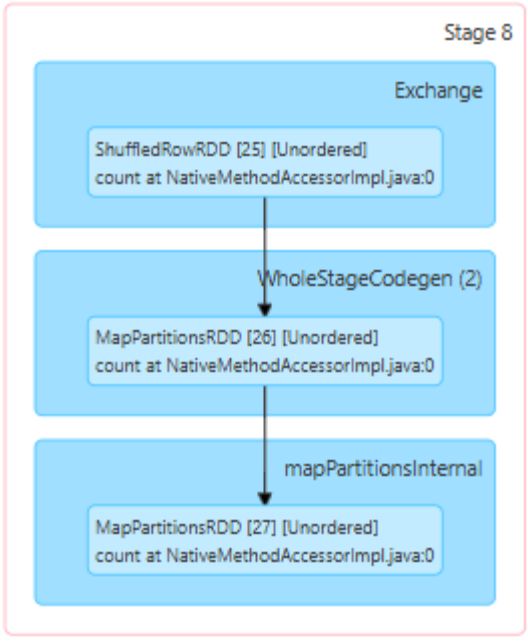
4. Dependency Type (Wide Dependency): The shuffle introduces wide dependencies, where each partition depends on data from multiple other partitions. However, the AQE optimization reduces the costs typically associated with such dependencies by optimizing partition sizes dynamically.

5. Final Processing (mapPartitionsInternal): The stage concludes with mapPartitionsInternal, which executes the final partition-level computations efficiently. This step ensures that all tasks within each partition are completed before proceeding to the next stage.



Explanation: Stage 6

1. Scan CSV (FileScanRDD): The stage begins by reading the CSV file using FileScanRDD. This operation loads the data in parallel across the cluster, allowing for efficient distributed data access.
2. Intermediate Steps (MapPartitionsRDD): After loading the data, it is processed and partitioned using MapPartitionsRDD. This step organizes the data into manageable partitions for further transformations and computations.
3. Optimization Techniques (WholeStageCodegen): Spark employs WholeStageCodegen to optimize the transformations in this stage. By combining multiple operations into a single execution plan, Spark minimizes intermediate overhead and improves the performance of this stage.
4. Dependency Type (Narrow Dependency): The operations in this stage involve narrow dependencies, where each partition processes its data independently. This eliminates the need for shuffles, ensuring efficient parallel processing.
5. Final Processing (Exchange): The stage concludes with an Exchange operation, redistributing or re-partitioning the data as required for subsequent stages. This step prepares the data for further computations or aligns it with the requirements of the next stage.



Explanation: Stage 8

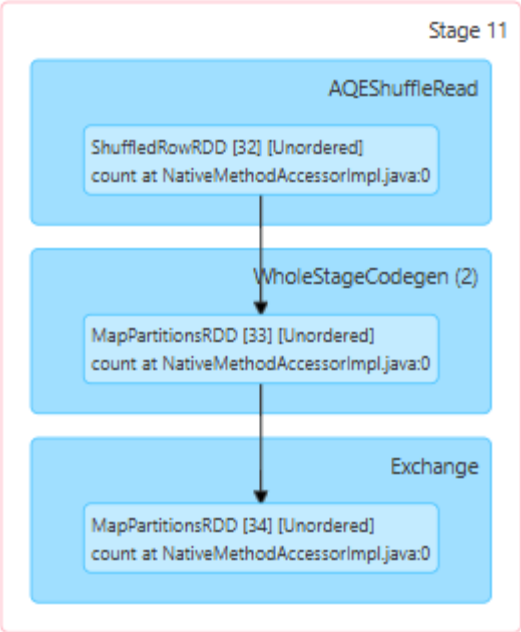
1. Exchange (ShuffledRowRDD): The stage begins with an Exchange operation using ShuffledRowRDD. This involves shuffling the data across partitions, typically required for operations like aggregations, joins, or sorts. The shuffle introduces wide dependencies, where partitions depend on data from multiple other partitions.

2. Intermediate Steps (MapPartitionsRDD): After the shuffle, MapPartitionsRDD is used to process and reorganize the data within partitions. This step prepares the shuffled data for further transformations.

3. Optimization Techniques (WholeStageCodegen): Spark applies WholeStageCodegen to optimize this stage by fusing multiple transformations into a single execution plan. This reduces overhead from intermediate steps, enhancing performance and reducing runtime.

4. Dependency Type (Wide Dependency): The shuffle operation creates wide dependencies, as data needs to be exchanged across partitions. These dependencies increase complexity, but they are necessary for global operations on the dataset.

5. Final Processing (mapPartitionsInternal): The stage concludes with mapPartitionsInternal, where partition-level computations are performed efficiently. This step ensures all operations within each partition are completed before the next stage begins.



Explanation: Stage 11

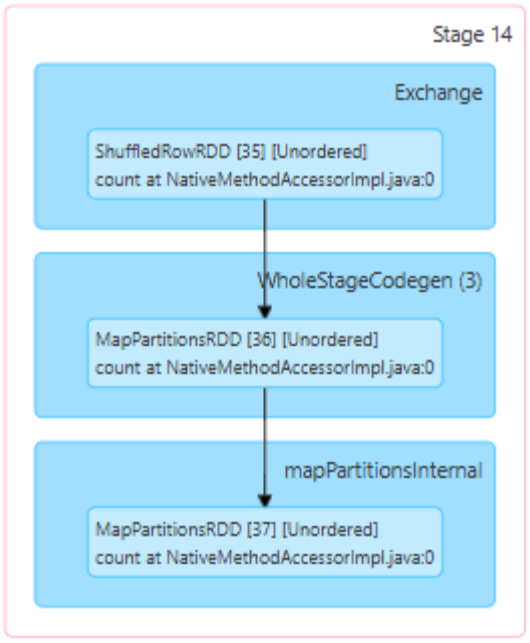
1. AQEShuffleRead (ShuffledRowRDD): This stage starts with an AQEShuffleRead operation utilizing ShuffledRowRDD. Adaptive Query Execution (AQE) dynamically optimizes the shuffle read process by adjusting partition sizes based on runtime statistics. This reduces resource usage and improves the efficiency of wide dependency operations such as joins or aggregations.

2. Intermediate Steps (MapPartitionsRDD): After the shuffle read, the data is partitioned and processed using MapPartitionsRDD. This step organizes the data into manageable chunks for downstream operations.

3. Optimization Techniques (WholeStageCodegen): Spark applies WholeStageCodegen to combine multiple transformations into a single execution plan. This optimization minimizes intermediate overhead, improves processing speed, and ensures efficient execution.

4. Dependency Type (Wide Dependency): The shuffle operation introduces wide dependencies, where data is exchanged across partitions. AQE optimizations mitigate the performance cost by dynamically managing partition sizes during execution.

5. Final Processing (Exchange): The stage concludes with an Exchange operation, redistributing or re-partitioning the data to prepare it for subsequent computations. This ensures the data is structured appropriately for the next stage.



Explanation: Stage 14

1. Exchange (ShuffledRowRDD): The stage starts with an Exchange operation using ShuffledRowRDD. This involves shuffling the data across partitions, which is necessary for operations like joins, aggregations, or sorts. The shuffle introduces wide dependencies, where partitions depend on data from multiple other partitions.

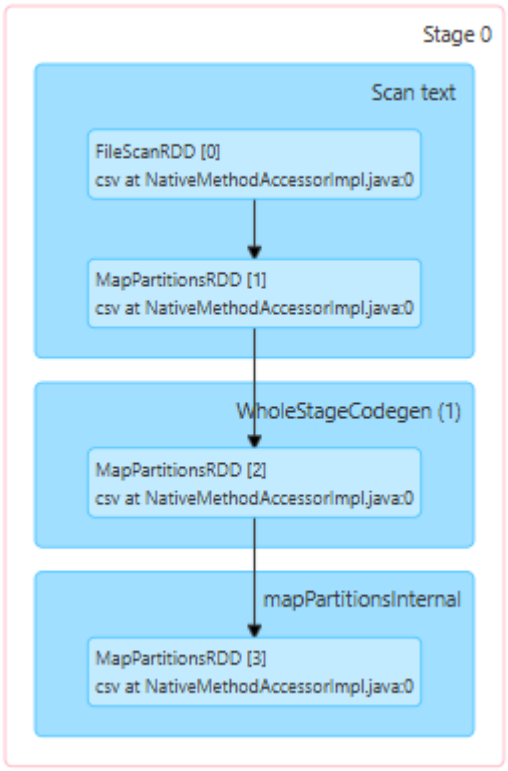
2. Intermediate Steps (MapPartitionsRDD): Following the shuffle, MapPartitionsRDD processes the reorganized data within each partition. This step prepares the data for subsequent transformations.

3. Optimization Techniques (WholeStageCodegen): Spark applies WholeStageCodegen to optimize this stage. By fusing multiple transformations into a single execution plan, Spark reduces the overhead from intermediate steps and improves performance.

4. Dependency Type (Wide Dependency): The shuffle operation creates wide dependencies, requiring data exchanges between partitions. This increases complexity and resource usage but is essential for global operations on the dataset.

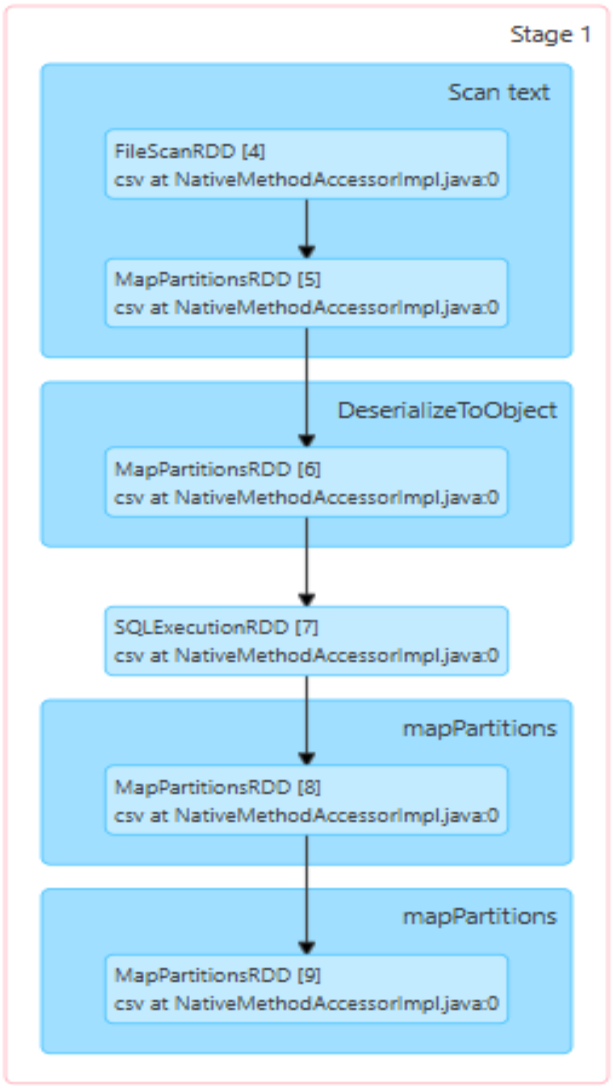
5. Final Processing (mapPartitionsInternal): The stage concludes with mapPartitionsInternal, where partition-level computations are executed efficiently. This ensures all transformations within each partition are completed before proceeding to the next stage.

STEP 2



Explanation: Stage 0

1. Scan Text (FileScanRDD): This stage begins with reading the CSV file as text using FileScanRDD. The operation is parallelized across the cluster to load data efficiently for processing.
2. Intermediate Steps (MapPartitionsRDD): After the data is read, it is partitioned and processed using MapPartitionsRDD. Each partition processes a chunk of the data independently, preparing it for further transformations.
3. Optimization Techniques (WholeStageCodegen): Spark employs WholeStageCodegen to optimize this stage by merging multiple transformations into a single execution plan. This reduces the runtime overhead caused by intermediate operations and ensures efficient execution.
4. Dependency Type (Narrow Dependency): The transformations involve narrow dependencies, meaning that each partition processes its data independently without requiring data exchanges or shuffles. This maximizes parallelism and minimizes execution costs.
5. Final Processing (mapPartitionsInternal): The stage concludes with mapPartitionsInternal, which performs partition-level computations efficiently. This step ensures the completion of all required transformations in this stage before moving to the next one.



Stage 1 Explanation

Scan Text (FileScanRDD):

The stage starts with a FileScanRDD operation, which reads the raw text data from a CSV source. This step loads the input data into Spark memory, preparing it for subsequent transformations.

Intermediate Steps (DeserializeToObject - MapPartitionsRDD):

Using MapPartitionsRDD, the raw text data is deserialized into structured objects, such as rows or records. This step ensures the data is organized and formatted for advanced processing.

SQL Execution (SQLExecutionRDD):

The SQLExecutionRDD executes transformations defined through SQL queries or DataFrame operations. It applies logical processing and prepares the data for fine-grained partition-based computations.

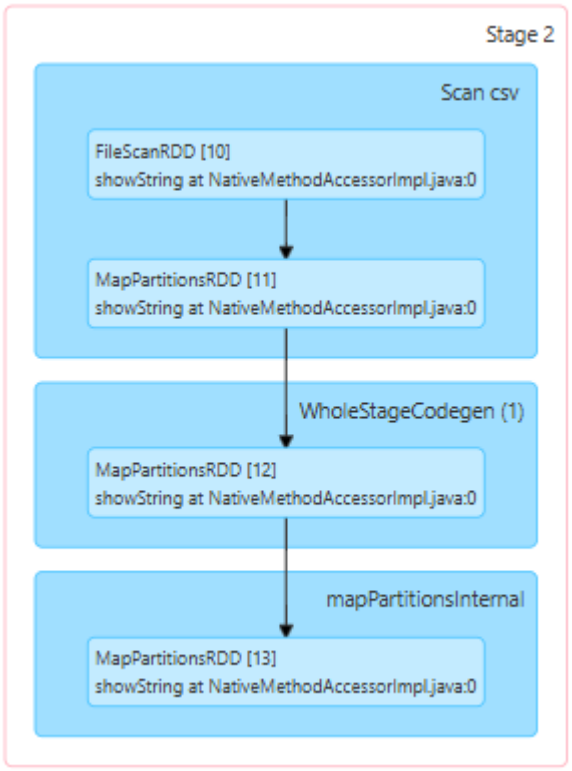
Optimization Techniques (mapPartitions - MapPartitionsRDD):

Partition-level transformations are applied iteratively using MapPartitionsRDD. These steps refine and process the data within each partition to optimize performance and reduce overhead.

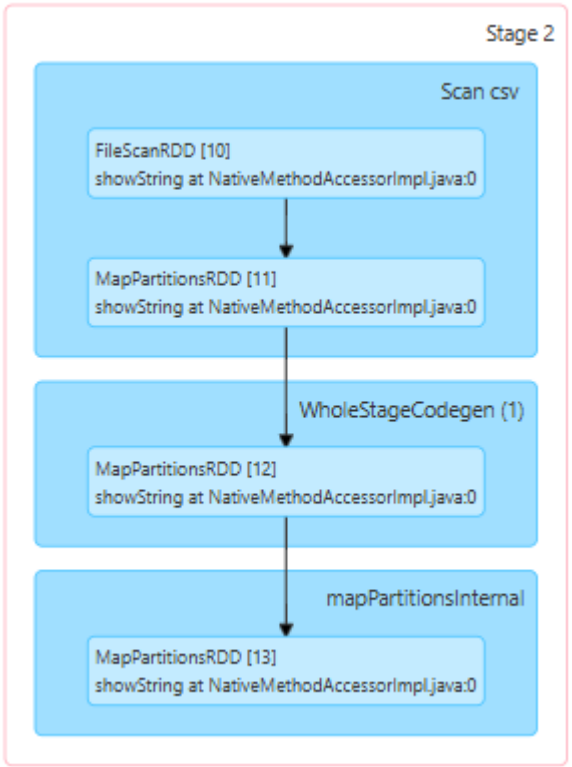
Final Processing (mapPartitions):

The stage concludes with additional MapPartitionsRDD operations to ensure all transformations are executed

efficiently within each partition. This prepares the data for use in the next stage.



Stage 2 Explanation



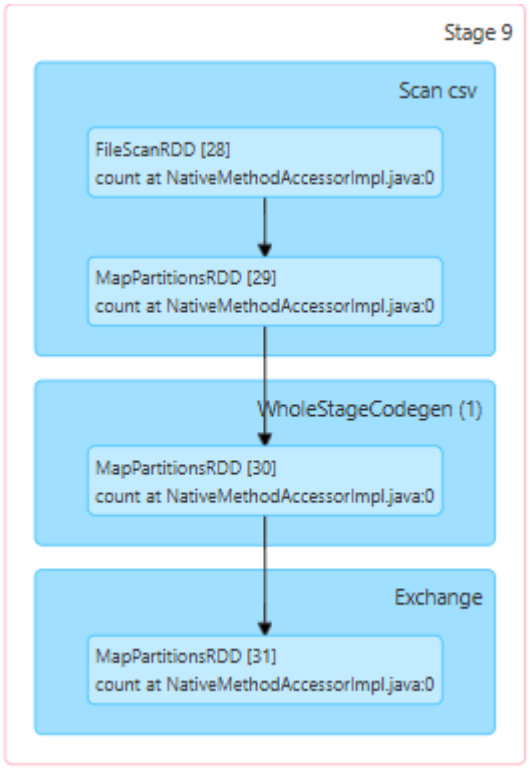
Scan CSV (FileScanRDD):
The stage begins with a FileScanRDD operation that reads data from a CSV file. This loads the raw data into Spark memory for further processing.

Intermediate Steps (MapPartitionsRDD):
A MapPartitionsRDD processes the scanned data within partitions. This step ensures the data is appropriately structured for subsequent transformations and optimizations.

Optimization Techniques (WholeStageCodegen):
Spark applies WholeStageCodegen to optimize this stage. This technique combines multiple transformations into a single execution plan, reducing overhead and enhancing performance.

Final Processing (mapPartitionsInternal):
The stage concludes with mapPartitionsInternal, where partition-level computations are executed efficiently. This ensures all transformations within partitions are finalized before moving to the next stage.

Stage 9 Explanation



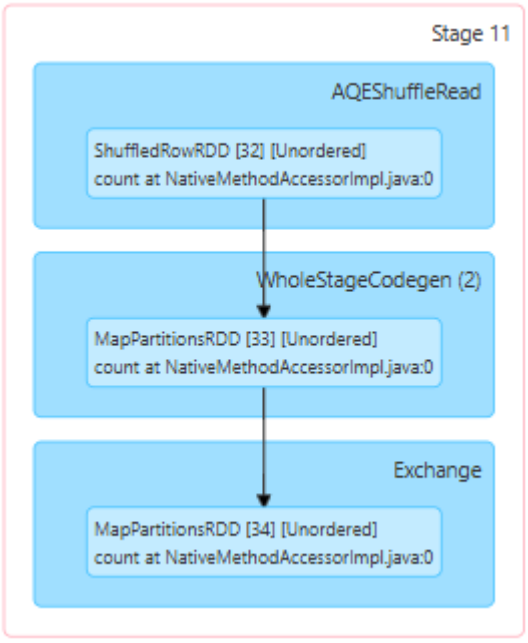
Scan CSV (FileScanRDD):
The stage starts with a FileScanRDD, where raw data from a CSV file is read and loaded into memory. This initializes the data processing pipeline.

Intermediate Steps (MapPartitionsRDD):
A MapPartitionsRDD processes the scanned data, performing operations like counting or preparing it for optimization. This step ensures the data is partitioned and organized efficiently.

Optimization Techniques (WholeStageCodegen):
Spark applies WholeStageCodegen to optimize transformations in this stage. By fusing multiple operations into a single execution plan, Spark reduces intermediate overhead and boosts performance.

Final Processing (Exchange):
The stage concludes with an Exchange operation that involves data shuffling across partitions. This step introduces wide dependencies and prepares the data for global computations in subsequent stages.

Stage 11 Explanation



AQEShuffleRead (ShuffledRowRDD):

The stage begins with an Adaptive Query Execution (AQE) ShuffleRead, where data from a previous shuffle operation is read into partitions. This step enables Spark to dynamically optimize partition sizes and balance workloads.

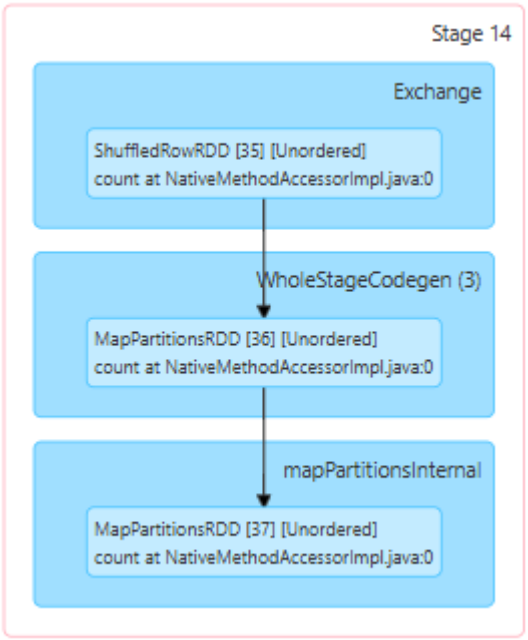
Intermediate Steps (WholeStageCodegen):

Spark applies WholeStageCodegen to optimize the execution of transformations. Multiple transformations are fused into a single execution plan, minimizing overhead and improving efficiency.

Final Processing (Exchange):

The stage concludes with an Exchange operation, where data is shuffled across partitions for further global computations. This introduces wide dependencies but prepares the dataset for subsequent processing.

Stage 14



Exchange (ShuffledRowRDD):

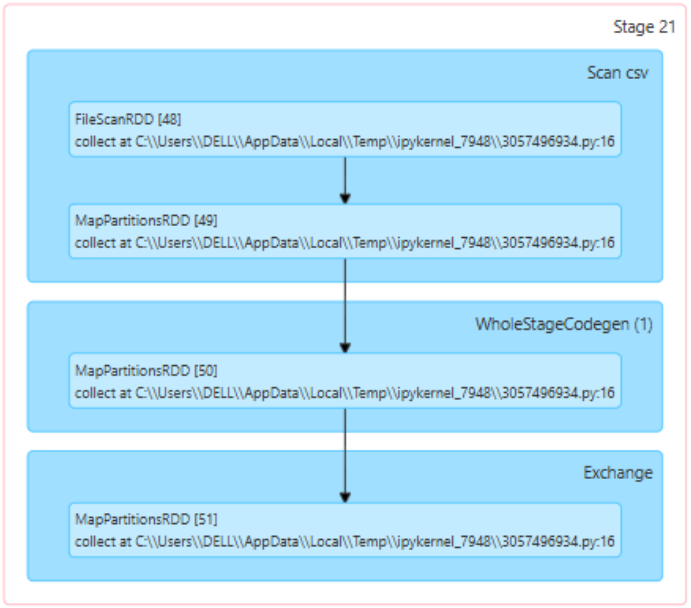
The stage begins with an Exchange operation using ShuffledRowRDD, where data is shuffled across partitions. This step ensures data is redistributed for operations like joins or aggregations, introducing wide dependencies.

Optimization Techniques (WholeStageCodegen):

Spark applies WholeStageCodegen to optimize transformations. This optimization fuses multiple operations into a single execution plan, reducing execution overhead and improving performance.

Final Processing (mapPartitionsInternal):

The stage concludes with mapPartitionsInternal, where partition-level computations are performed efficiently. This ensures all required transformations within partitions are completed before moving to the next stage.



Stage 21 Explanation

Scan CSV (FileScanRDD):

The stage begins with a FileScanRDD operation, reading data from a CSV file. This step loads raw data into Spark memory, serving as the starting point for the processing pipeline.

Intermediate Steps (MapPartitionsRDD):

A MapPartitionsRDD processes the scanned data within each partition. This ensures the data is appropriately structured and organized for downstream transformations.

Optimization Techniques (WholeStageCodegen):

Spark applies WholeStageCodegen to optimize the execution of transformations. By combining multiple operations into a single execution plan, Spark reduces execution overhead and improves efficiency.

Final Processing (Exchange):

The stage concludes with an Exchange operation, where data is shuffled across partitions. This step ensures the dataset is redistributed for operations like joins, aggregations, or other global transformations.

Stage 23

AQEShuffleRead (ShuffledRowRDD):

The stage starts with an Adaptive Query Execution (AQE) ShuffleRead operation using ShuffledRowRDD. This step reads data from a previous shuffle operation, dynamically optimizing partition sizes to balance workload and improve efficiency.

Dependency Type:

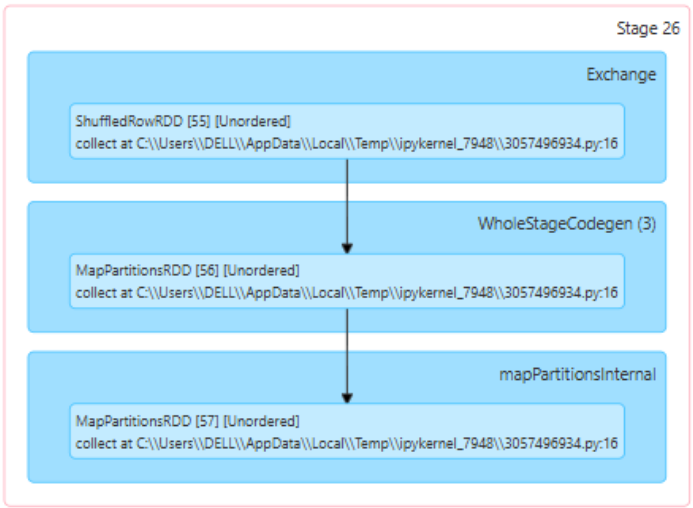
This step involves wide dependencies, as partitions depend on data from multiple other partitions. This increases complexity but is necessary for global data transformations.

Optimization Techniques (WholeStageCodegen):

Spark applies WholeStageCodegen to combine multiple transformations into a single execution plan. This optimization reduces intermediate overhead and speeds up the computation.

Final Processing (Exchange):

The stage concludes with an Exchange operation that redistributes data across partitions. This step ensures the data is prepared for global operations like joins or aggregations.



Stage 26 Explanation

Exchange (ShuffledRowRDD):

The stage begins with an Exchange operation using ShuffledRowRDD, where data is shuffled across partitions. This redistribution is essential for global operations like joins or aggregations.

Dependency Type:

This step involves wide dependencies, as data in each partition relies on multiple other partitions. This increases computational complexity but is necessary for accurate results in global transformations.

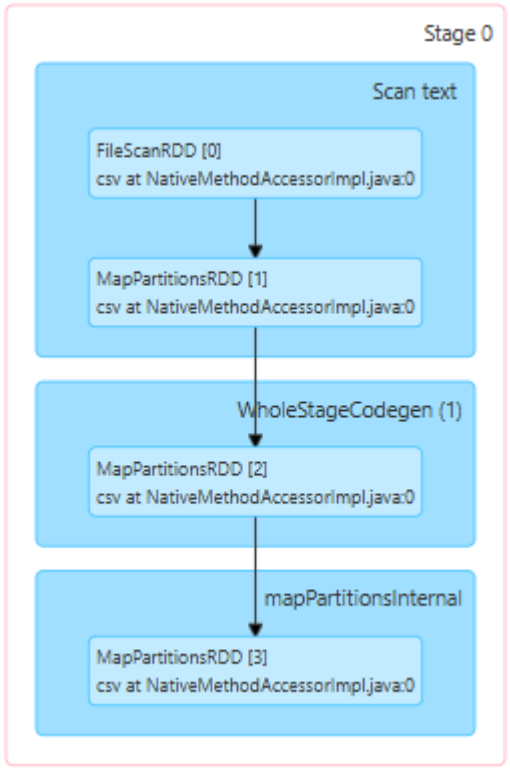
Optimization Techniques (WholeStageCodegen):

Spark applies WholeStageCodegen to optimize transformations by fusing multiple operations into a single execution plan. This reduces the overhead of intermediate steps and improves performance.

Final Processing (mapPartitionsInternal):

The stage concludes with mapPartitionsInternal, performing efficient partition-level computations. This ensures all transformations within each partition are completed before moving to the next stage.

STEP 3



Stage 0 Explanation

Scan Text (FileScanRDD):

The stage begins with a FileScanRDD operation, where raw text data (e.g., CSV) is read from the input source. This operation loads the data into memory for subsequent processing.

Intermediate Steps (MapPartitionsRDD):

A MapPartitionsRDD processes the scanned data within each partition. This step organizes the data for transformations while ensuring efficient partition-level operations.

Dependency Type:

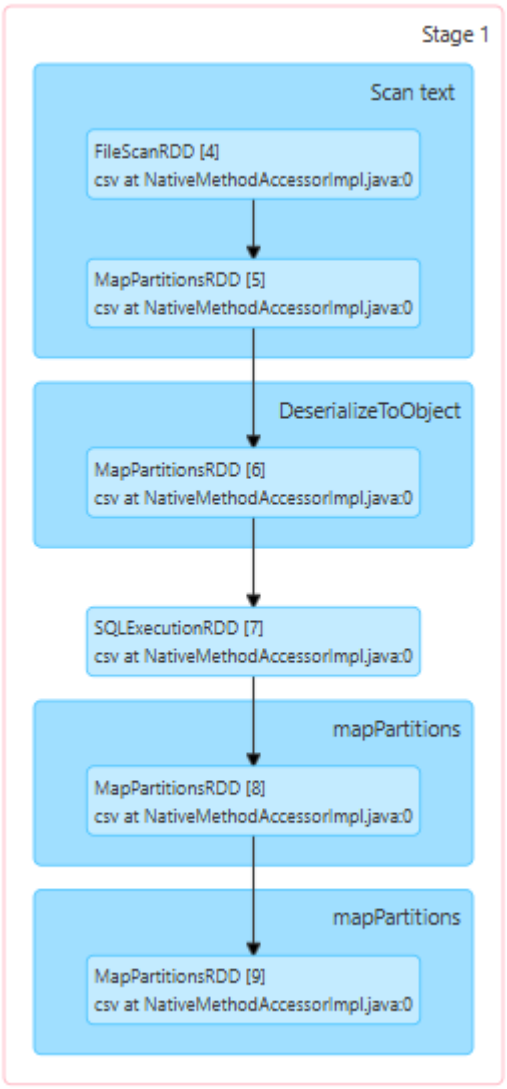
This stage involves narrow dependencies, where partitions depend only on a subset of the data. This ensures efficient local transformations without the need for shuffling.

Optimization Techniques (WholeStageCodegen):

Spark applies WholeStageCodegen to optimize the execution of transformations by fusing multiple operations into a single plan. This reduces intermediate overhead and improves processing speed.

Final Processing (mapPartitionsInternal):

The stage concludes with mapPartitionsInternal, which efficiently completes the computations within each partition before transitioning to the next stage.



Stage 1 Explanation

Scan Text (FileScanRDD):

The stage starts with a FileScanRDD, which reads raw text data (e.g., CSV) from the input source. This step loads the data into memory for further transformations.

Intermediate Steps (DeserializeToObject):

Using MapPartitionsRDD, the raw text is deserialized into structured objects such as rows or records. This prepares the data for SQL-based or logical transformations.

SQL Execution (SQLExecutionRDD):

The SQLExecutionRDD applies SQL transformations or DataFrame operations on the structured data. This enables advanced computations and logical processing.

Partition-Level Refinements (MapPartitionsRDD):

Successive MapPartitionsRDD steps are executed to perform partition-level transformations. These steps refine the data for downstream processing and improve local computation efficiency.

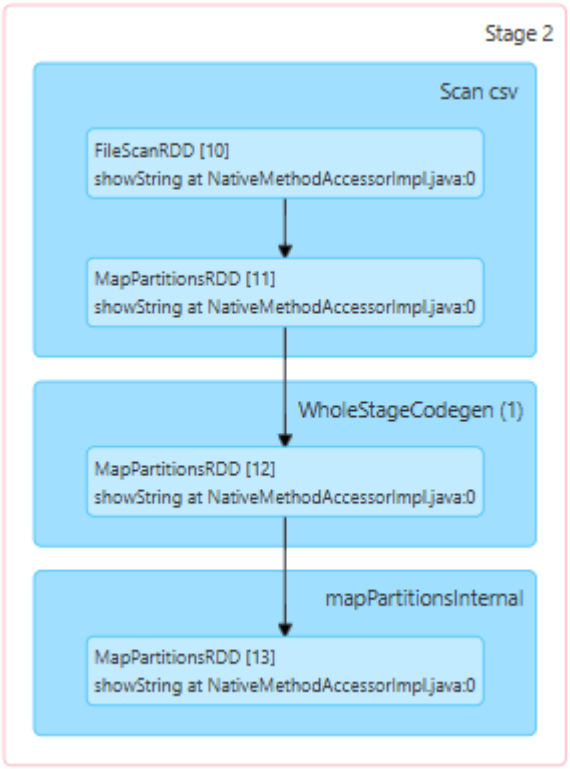
Dependency Type:

This stage primarily involves narrow dependencies, where each partition works on its own subset of the data without requiring data from other partitions.

Final Processing (MapPartitions):

The stage concludes with final MapPartitionsRDD steps, completing the required computations efficiently within each partition before transitioning to the next stage.

Stage 2 Explanation



Scan CSV (FileScanRDD):

The stage starts with a FileScanRDD operation that reads data from a CSV file. This initial step loads the raw data into memory for further processing.

Intermediate Steps (MapPartitionsRDD):

A MapPartitionsRDD processes the scanned data within each partition. This transformation organizes the data, ensuring it's ready for optimized execution.

Dependency Type:

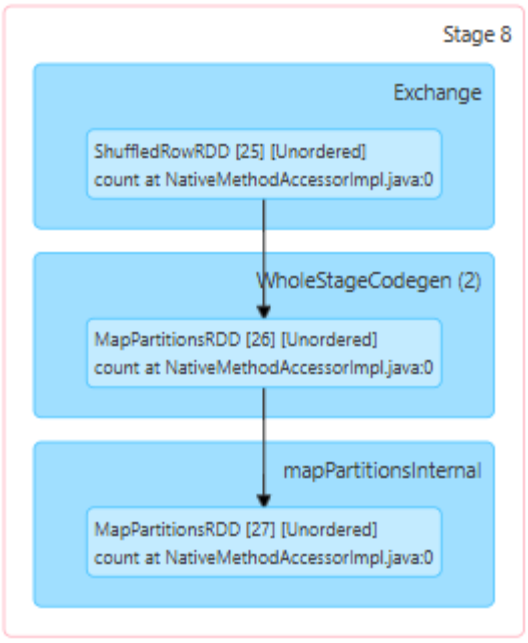
This stage involves narrow dependencies, where each partition operates independently on a subset of the data without requiring shuffling or communication with other partitions.

Optimization Techniques (WholeStageCodegen):

Spark applies WholeStageCodegen to optimize execution. This combines multiple transformations into a single execution plan, reducing overhead and improving computational efficiency.

Final Processing (mapPartitionsInternal):

The stage concludes with mapPartitionsInternal, where partition-level computations are executed efficiently. This step ensures that all transformations within each partition are completed before moving to the next stage.



Stage 8 Explanation

Exchange (ShuffledRowRDD):

The stage begins with an Exchange operation using ShuffledRowRDD. This involves shuffling data across partitions, redistributing it to support global operations such as joins or aggregations.

Dependency Type:

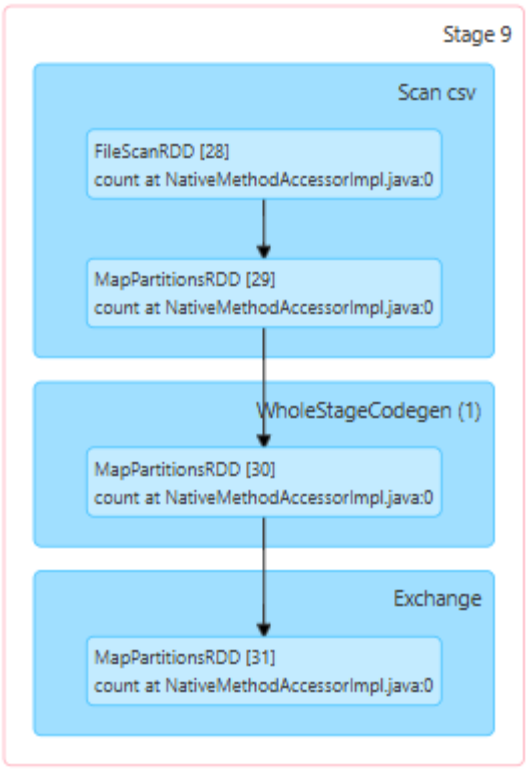
This step introduces wide dependencies, where each partition relies on data from multiple other partitions. This increases complexity but is essential for achieving accurate global transformations.

Optimization Techniques (WholeStageCodegen):

Spark applies WholeStageCodegen to fuse multiple transformations into a single execution plan. This optimization reduces intermediate overhead, making the execution more efficient.

Final Processing (mapPartitionsInternal):

The stage concludes with mapPartitionsInternal, where computations within each partition are executed efficiently. This step ensures that all transformations within the partitions are completed before moving to the next stage.



Stage 9 Explanation

Scan CSV (FileScanRDD):

The stage begins with a FileScanRDD operation, which reads data from a CSV file. This is the starting point for loading raw data into memory for further transformations.

Intermediate Steps (MapPartitionsRDD):

The data is processed within partitions using a MapPartitionsRDD. This ensures the data is structured and prepared for optimized execution in subsequent steps.

Dependency Type:

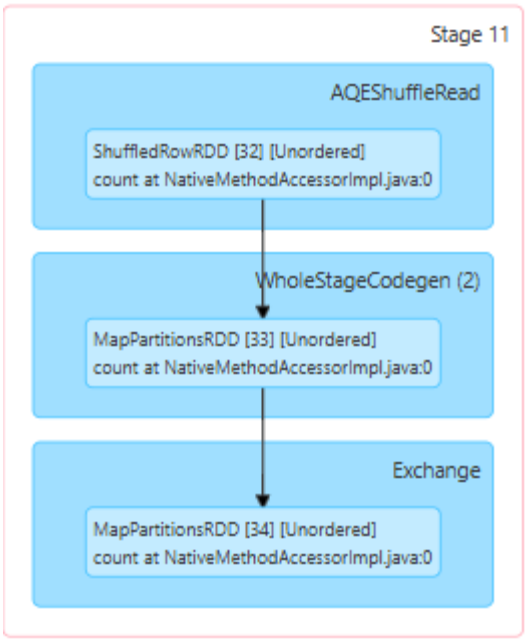
This stage primarily involves narrow dependencies, where each partition works independently on a subset of the data without requiring shuffling.

Optimization Techniques (WholeStageCodegen):

Spark applies WholeStageCodegen to optimize the execution of transformations. By combining multiple operations into a single execution plan, this reduces intermediate overhead and improves processing efficiency.

Final Processing (Exchange):

The stage concludes with an Exchange operation, where data is shuffled across partitions. This step prepares the dataset for global operations, such as joins or aggregations, introducing wide dependencies in preparation for the next stage.



Stage 11 Explanation

AQEShuffleRead (ShuffledRowRDD):

The stage begins with an Adaptive Query Execution (AQE) ShuffleRead operation using ShuffledRowRDD. This step reads shuffled data from a previous stage and dynamically optimizes partition sizes for balanced processing.

Dependency Type:

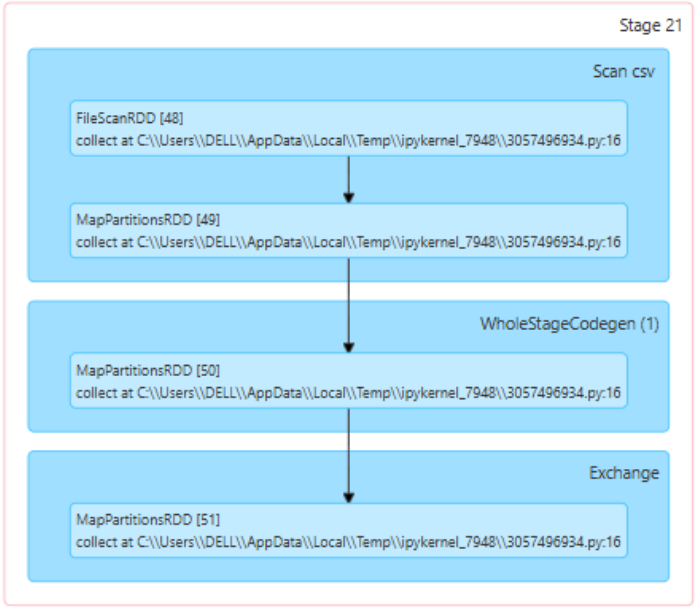
This operation involves wide dependencies, where each partition relies on data from multiple other partitions, increasing complexity but enabling global transformations.

Optimization Techniques (WholeStageCodegen):

Spark applies WholeStageCodegen to optimize transformations by fusing multiple operations into a single execution plan. This minimizes intermediate overhead and accelerates computation.

Final Processing (Exchange):

The stage concludes with an Exchange operation, where data is reshuffled across partitions. This step prepares the dataset for further global operations, such as joins or aggregations, ensuring the necessary data redistribution for the next stage.



Stage 21 Explanation

Scan CSV (FileScanRDD):

The stage begins with a FileScanRDD operation, reading raw data from a CSV file. This serves as the initial step for loading data into memory for further processing.

Intermediate Steps (MapPartitionsRDD):

The data is processed using a MapPartitionsRDD, ensuring it is partitioned and prepared for efficient computation during subsequent transformations.

Dependency Type:

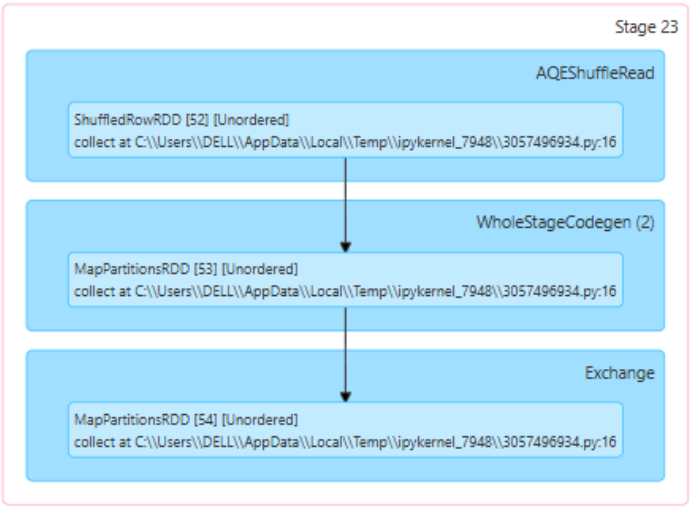
This stage involves narrow dependencies, as each partition operates independently without requiring data from other partitions, ensuring localized and efficient execution.

Optimization Techniques (WholeStageCodegen):

Spark applies WholeStageCodegen to fuse multiple transformations into a single execution plan. This optimization reduces the overhead associated with intermediate steps and enhances processing speed.

Final Processing (Exchange):

The stage concludes with an Exchange operation, redistributing data across partitions to enable global transformations like joins or aggregations. This introduces wide dependencies, ensuring data is correctly organized for the next stage.



Stage 23 Explanation

AQEShuffleRead (ShuffledRowRDD):

The stage begins with an Adaptive Query Execution (AQE) ShuffleRead operation, where shuffled data from a previous stage is read. This dynamically adjusts partition sizes to optimize workload balance and improve efficiency.

Intermediate Steps (MapPartitionsRDD):

A MapPartitionsRDD processes the shuffled data, ensuring it is organized and prepared for optimized execution in subsequent steps.

Dependency Type:

This operation involves wide dependencies, as data in each partition depends on multiple other partitions, making it essential for global data transformations.

Optimization Techniques (WholeStageCodegen):

Spark applies WholeStageCodegen to fuse multiple transformations into a single execution plan. This optimization reduces intermediate overhead and accelerates computation.

Final Processing (Exchange):

The stage concludes with an Exchange operation that redistributes data across partitions. This prepares the dataset for further global transformations, such as joins or aggregations, and sets up the next stage of execution.

Stage 26 Explanation

Exchange (ShuffledRowRDD):

The stage begins with an Exchange operation using ShuffledRowRDD. This redistributes data across partitions, enabling global operations such as joins, aggregations, or sorts by organizing data across partitions.

Intermediate Steps (MapPartitionsRDD):

The redistributed data is processed using a MapPartitionsRDD. This step ensures data is properly structured and prepared for further optimized transformations.

Dependency Type:

This operation involves wide dependencies, as partitions rely on data from multiple other partitions, which increases complexity but is necessary for global transformations.

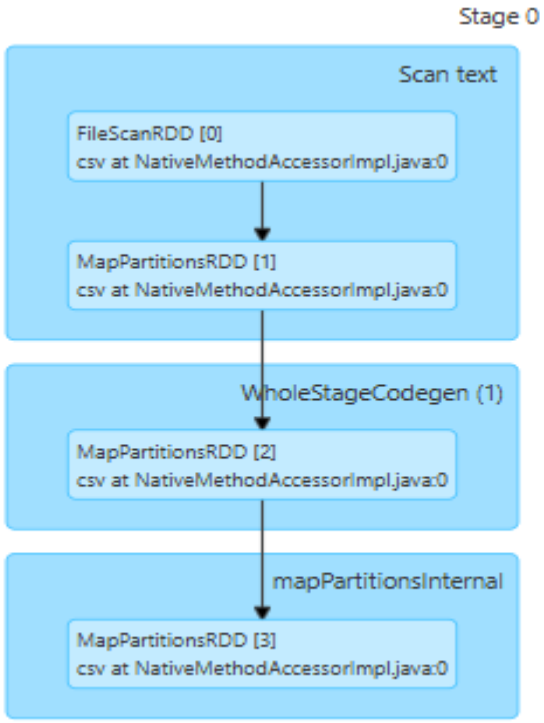
Optimization Techniques (WholeStageCodegen):

Spark applies WholeStageCodegen to fuse multiple transformations into a single execution plan. This optimization minimizes intermediate overhead and enhances execution efficiency.

Final Processing (mapPartitionsInternal):

The stage concludes with mapPartitionsInternal, where final computations are performed efficiently within each partition. This ensures that all transformations are completed before transitioning to the next stage.

STEP4 A



Stage 0 Explanation

Scan Text (FileScanRDD):

The stage starts with a FileScanRDD operation, which reads raw text data (e.g., CSV files) from the input source. This step loads the data into memory to initialize the processing pipeline.

Intermediate Steps (MapPartitionsRDD):

The loaded data is processed using a MapPartitionsRDD, partitioning it to ensure efficient computation in subsequent steps. This operation organizes the data for better downstream processing.

Dependency Type:

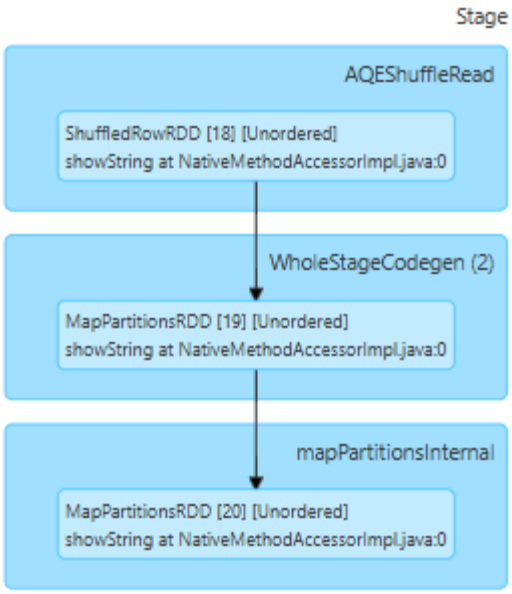
This stage involves narrow dependencies, where each partition processes only its subset of data without requiring communication with other partitions.

Optimization Techniques (WholeStageCodegen):

Spark applies WholeStageCodegen to combine multiple transformations into a single execution plan, reducing intermediate steps and improving computational efficiency.

Final Processing (mapPartitionsInternal):

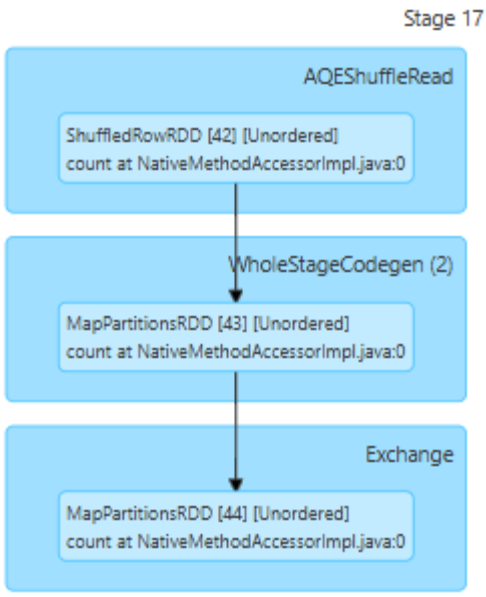
The stage concludes with mapPartitionsInternal, which performs efficient computations within each partition. This ensures that all necessary transformations are complete before transitioning to the next stage.



Stage 5 Explanation

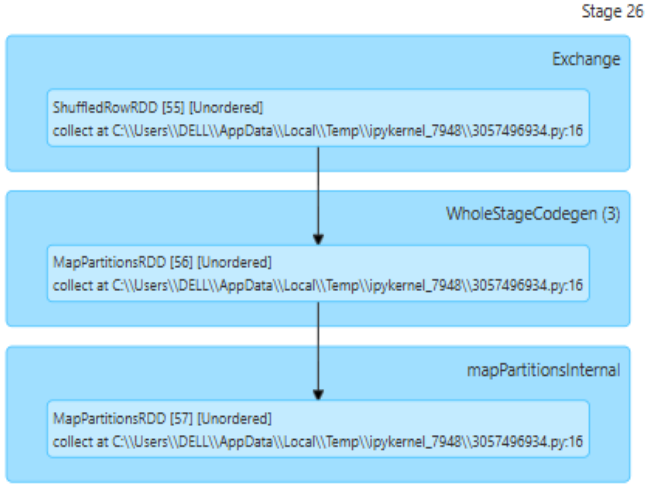
AQEShuffleRead (ShuffledRowRDD):
The stage begins with an Adaptive Query Execution (AQE) ShuffleRead operation using ShuffledRowRDD. This reads shuffled data from a prior stage and dynamically optimizes partition sizes to ensure balanced workload and efficient processing.
Intermediate Steps (MapPartitionsRDD):
The shuffled data is processed using a MapPartitionsRDD, which structures the data to prepare it for further transformations. This step optimizes the data organization at the partition level.

Dependency Type:
This stage involves wide dependencies, as partitions depend on data from multiple other partitions, which is necessary for global transformations.
Optimization Techniques (WholeStageCodegen):
Spark applies WholeStageCodegen to merge multiple operations into a single execution plan. This reduces intermediate overhead and speeds up computation.
Final Processing (mapPartitionsInternal):
The stage concludes with mapPartitionsInternal, where partition-level computations are completed efficiently. This ensures all necessary transformations within partitions are finalized before transitioning to the next stage.



Stage 17 Explanation

AQEShuffleRead (ShuffledRowRDD):
The stage begins with an Adaptive Query Execution (AQE) ShuffleRead operation using ShuffledRowRDD. This step reads shuffled data from previous stages and adjusts partition sizes dynamically for workload balance and efficient processing.
Intermediate Steps (MapPartitionsRDD):
A MapPartitionsRDD processes the shuffled data, ensuring it is structured and optimized for subsequent transformations and computations.
Dependency Type:
This stage involves wide dependencies, where partitions rely on data from multiple other partitions. This is critical for global transformations like joins or aggregations.
Optimization Techniques (WholeStageCodegen):
Spark applies WholeStageCodegen to fuse multiple transformations into a single execution plan. This optimization reduces intermediate overhead and improves computational efficiency.
Final Processing (Exchange):
The stage concludes with an Exchange operation, redistributing data across partitions. This prepares the dataset for further global computations in subsequent stages, enabling effective aggregation or join operations.



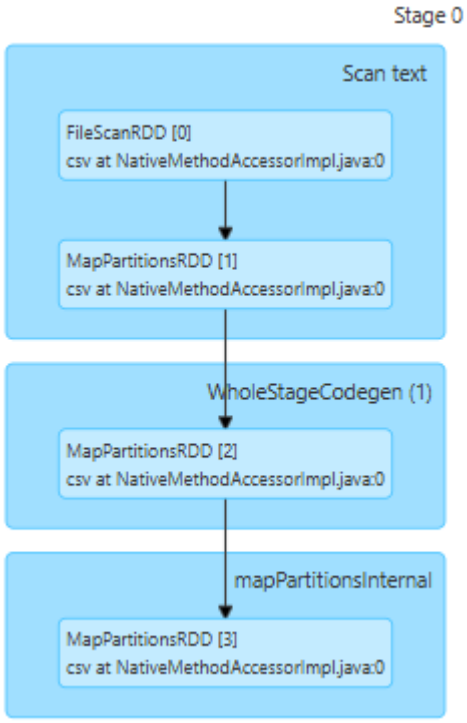
Stage 26 Explanation

Exchange (ShuffledRowRDD):
The stage starts with an Exchange operation using ShuffledRowRDD, where data is shuffled across partitions to enable global transformations like joins, aggregations, or sorting. This redistributes data to ensure correct results for downstream computations.
Intermediate Steps (MapPartitionsRDD):
A MapPartitionsRDD processes the redistributed data, preparing it for further computations by optimizing the structure and organization of data at the partition level.
Dependency Type:
This stage involves wide dependencies, as partitions rely on data from multiple other partitions. This is essential for global data reorganization but adds computational complexity.
Optimization Techniques (WholeStageCodegen):
Spark applies WholeStageCodegen to combine multiple operations into a single execution plan, significantly reducing intermediate steps and improving performance.
Final Processing (mapPartitionsInternal):
The stage concludes with mapPartitionsInternal, where partition-level computations are performed efficiently. This step ensures that all transformations are complete within partitions before moving to the next stage.

Stage 29 Explanation

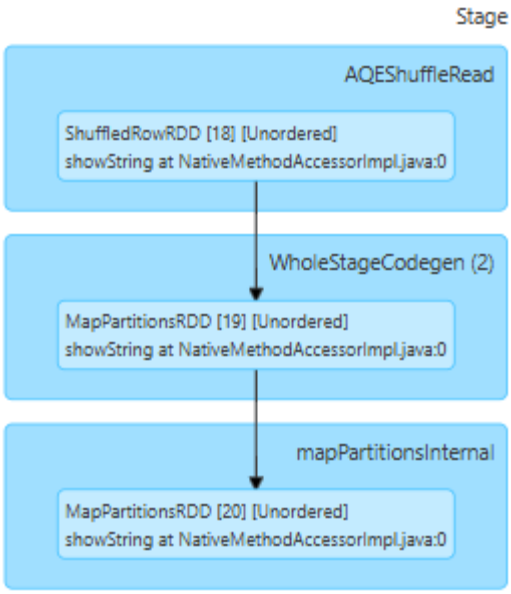
AQEShuffleRead (ShuffledRowRDD):
The stage begins with an Adaptive Query Execution (AQE) ShuffleRead operation using ShuffledRowRDD. This dynamically retrieves data shuffled from previous stages, optimizing partition sizes to balance the workload for efficient processing.
Intermediate Steps (MapPartitionsRDD):
A MapPartitionsRDD processes the shuffled data at the partition level. This ensures that the data is organized and structured for optimized execution in subsequent computations.
Dependency Type:
This stage involves wide dependencies, where partitions depend on data from multiple other partitions. This is crucial for global transformations but adds computational overhead.
Optimization Techniques (WholeStageCodegen):
Spark applies WholeStageCodegen to merge multiple transformations into a single execution plan, minimizing intermediate operations and improving overall performance.
Final Processing (mapPartitionsInternal):
The stage concludes with mapPartitionsInternal, where efficient computations are performed within each partition. This ensures that all necessary transformations are completed before transitioning to the next stage.

STEP 4 B



Stage 0 Explanation

Scan Text (FileScanRDD):
The stage begins with a FileScanRDD operation that reads raw text data (e.g., CSV files) from the input source. This is the first step to load the data into memory for processing.
Intermediate Steps (MapPartitionsRDD):
The data is processed using a MapPartitionsRDD, which partitions the data to ensure efficient transformations and computation in the subsequent stages.
Dependency Type:
This stage involves narrow dependencies, where each partition operates independently on its subset of the data without requiring communication between partitions.
Optimization Techniques (WholeStageCodegen):
Spark applies WholeStageCodegen to optimize execution by fusing multiple transformations into a single execution plan. This reduces intermediate steps and enhances the computational efficiency.
Final Processing (mapPartitionsInternal):
The stage concludes with mapPartitionsInternal, performing efficient computations within each partition. This step ensures that all necessary transformations are complete before transitioning to the next stage.



Stage 5 Explanation

AQEShuffleRead (ShuffledRowRDD):

The stage begins with an Adaptive Query Execution (AQE) ShuffleRead operation using ShuffledRowRDD. This operation dynamically reads shuffled data from previous stages and adjusts partition sizes for optimal workload balance.

Intermediate Steps (MapPartitionsRDD):

The shuffled data is processed using a MapPartitionsRDD, which organizes and prepares the data for subsequent transformations, ensuring efficient execution at the partition level.

Dependency Type:

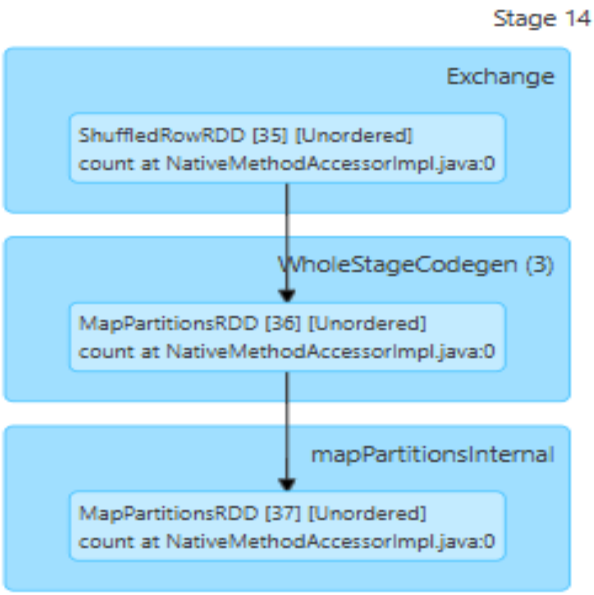
This stage involves wide dependencies, where data in each partition depends on multiple other partitions. This is essential for global transformations but adds computational complexity.

Optimization Techniques (WholeStageCodegen):

Spark applies WholeStageCodegen to merge multiple transformations into a single execution plan. This optimization reduces intermediate overhead, streamlines processing, and enhances performance.

Final Processing (mapPartitionsInternal):

The stage concludes with mapPartitionsInternal, where partition-level computations are completed efficiently. This ensures all necessary transformations are finalized before moving to the next stage.



Stage 14 Explanation

Exchange (ShuffledRowRDD):

The stage begins with an Exchange operation using ShuffledRowRDD. This operation redistributes data across partitions, enabling global transformations such as joins, aggregations, or sorting by reorganizing the data.

Intermediate Steps (MapPartitionsRDD):

The redistributed data is processed through a MapPartitionsRDD. This ensures the data is properly structured and ready for the next set of transformations.

Dependency Type:

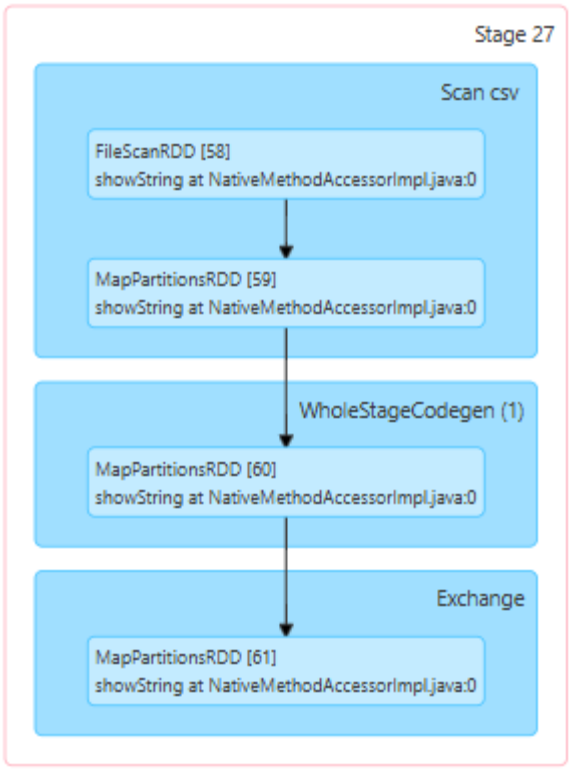
This stage involves wide dependencies, where data in each partition depends on multiple other partitions. This is critical for global operations but increases computational complexity.

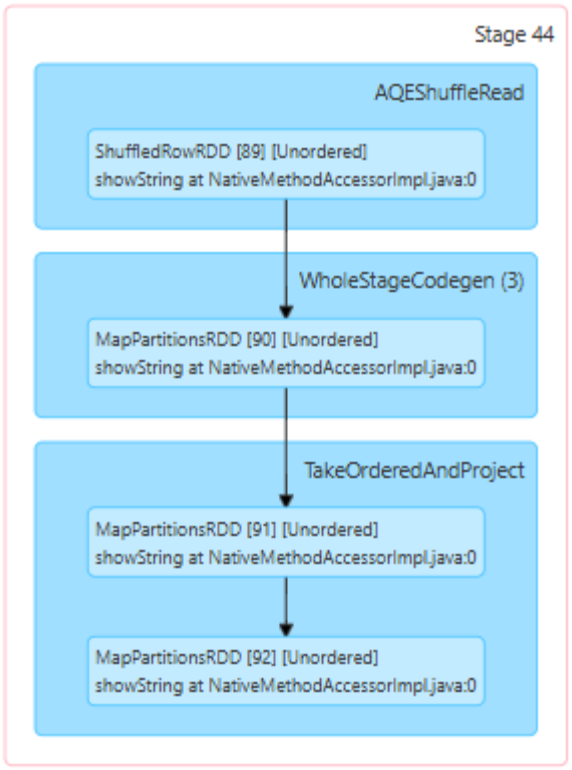
Optimization Techniques (WholeStageCodegen):

Spark applies WholeStageCodegen to optimize transformations by combining multiple operations into a single execution plan. This reduces the overhead of intermediate steps and enhances computational efficiency.

Final Processing (mapPartitionsInternal):

The stage concludes with mapPartitionsInternal, where efficient computations are performed within each partition. This step ensures that all necessary transformations are completed before moving to the next stage.





Stage 27 Explanation

Scan CSV (FileScanRDD):

The stage begins with a FileScanRDD operation, where raw data is read from a CSV file. This step loads the data into memory, setting the foundation for subsequent transformations.

Intermediate Steps (MapPartitionsRDD):

The loaded data is processed using a MapPartitionsRDD, which partitions and organizes the data. This ensures the data is ready for efficient computation in the next steps.

Dependency Type:

This stage involves narrow dependencies, where each partition processes its subset of data independently without requiring data from other partitions.

Optimization Techniques (WholeStageCodegen):

Spark applies WholeStageCodegen to fuse multiple operations into a single execution plan. This reduces the overhead of intermediate computations and improves performance.

Final Processing (Exchange):

The stage concludes with an Exchange operation, where data is shuffled across partitions. This introduces wide dependencies, redistributing the data to support global transformations like joins or aggregations in subsequent stages.

Stage 44 Explanation

AQEShuffleRead (ShuffledRowRDD):

The stage begins with an Adaptive Query Execution (AQE) ShuffleRead operation using ShuffledRowRDD. This operation reads shuffled data from previous stages and optimizes partition sizes dynamically for balanced processing and efficient resource utilization.

Intermediate Steps (MapPartitionsRDD):

A MapPartitionsRDD processes the shuffled data within partitions. This ensures the data is structured and prepared for efficient execution of subsequent transformations.

Dependency Type:

This stage involves wide dependencies, as partitions rely on data from multiple other partitions. This is necessary for global transformations, such as sorting or aggregations.

Optimization Techniques (WholeStageCodegen):

Spark applies WholeStageCodegen to optimize the execution by fusing multiple transformations into a single execution plan. This minimizes intermediate steps and enhances performance.

Final Processing (TakeOrderedAndProject):

The stage concludes with a TakeOrderedAndProject operation executed through MapPartitionsRDD. This step extracts the required number of ordered elements based on the transformation logic (e.g., top-k results) and finalizes the projection for downstream operations or output. This ensures the necessary computations are completed efficiently within partitions.

STEP 4 C

Stage 0 Explanation

Scan Text (FileScanRDD):

The stage begins with a FileScanRDD operation that reads raw text data (e.g., CSV) from the input source. This initializes the pipeline by loading the data into memory for processing.

Intermediate Steps (MapPartitionsRDD):

A MapPartitionsRDD processes the loaded data within each partition. This ensures the data is partitioned and organized efficiently for the next steps.

Dependency Type:

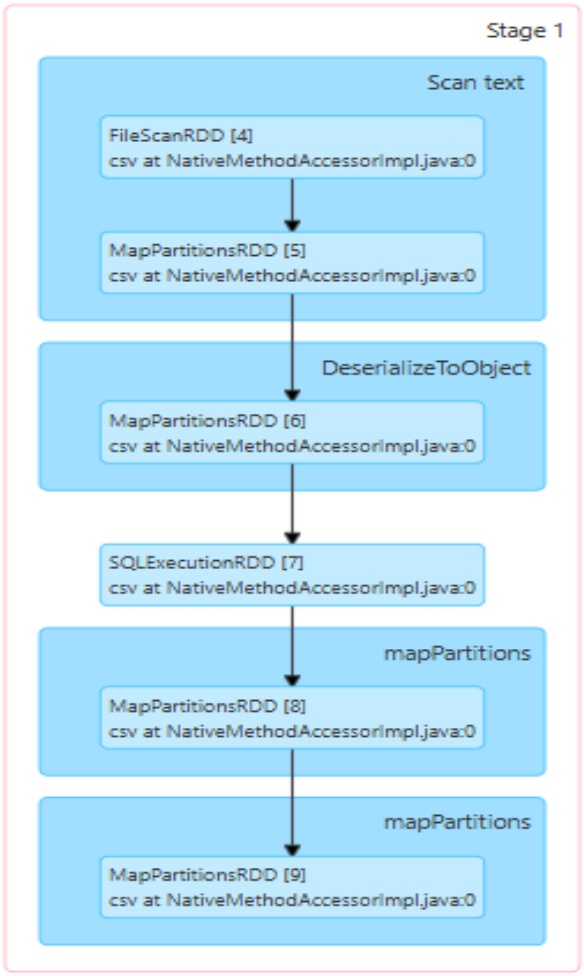
This stage involves narrow dependencies, where each partition operates independently on its subset of data, without requiring communication between partitions.

Optimization Techniques (WholeStageCodegen):

Spark applies WholeStageCodegen to fuse multiple operations into a single execution plan. This optimization reduces intermediate steps and enhances computational efficiency.

Final Processing (mapPartitionsInternal):

The stage concludes with mapPartitionsInternal, where efficient partition-level computations are executed. This ensures all transformations within each partition are completed before transitioning to the next stage.



Stage 1 Explanation

Scan Text (FileScanRDD):

The stage starts with a FileScanRDD operation, which reads raw text data (e.g., CSV files) from the input source. This step loads the data into memory, initializing the data processing pipeline.

Intermediate Steps (DeserializeToObject - MapPartitionsRDD):

The raw text data is deserialized into structured objects using a MapPartitionsRDD. This step converts the raw input into structured rows or records suitable for downstream transformations.

SQL Execution (SQLExecutionRDD):

The deserialized data is processed using SQLExecutionRDD, where SQL or DataFrame operations are applied. This layer is responsible for executing logical computations or transformations defined by the user.

Partition-Level Refinement (MapPartitionsRDD):

Subsequent MapPartitionsRDD operations perform localized computations within partitions. These steps refine the data and apply any additional transformations required before moving to the next stage.

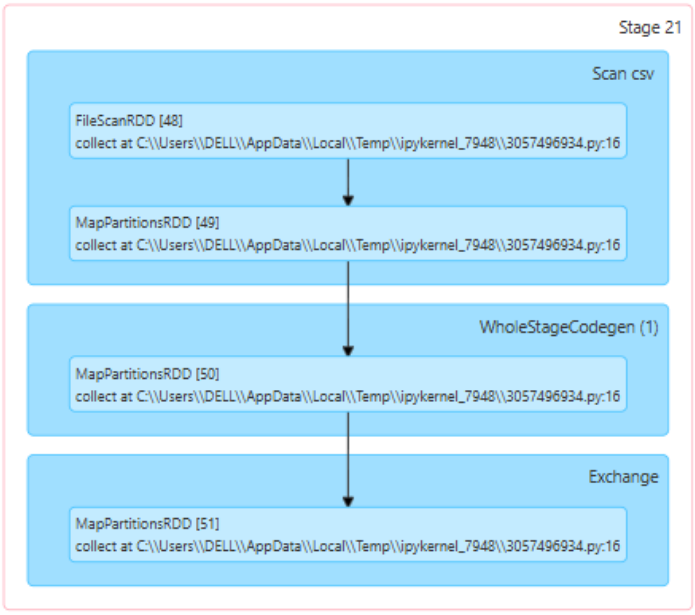
Dependency Type:

This stage primarily involves narrow dependencies, where each partition processes its subset of the data independently, ensuring efficient localized operations.

Final Processing (mapPartitions):

The stage concludes with partition-level computations, ensuring all transformations are completed efficiently

before the data transitions to the next stage.



Stage 21 Explanation

Scan CSV (FileScanRDD):

The stage begins with a FileScanRDD operation, where raw data is read from a CSV file. This is the initial step to load the dataset into memory for further processing.

Intermediate Steps (MapPartitionsRDD):

The scanned data is processed by a MapPartitionsRDD, partitioning and organizing the data for efficient computation in subsequent transformations.

Dependency Type:

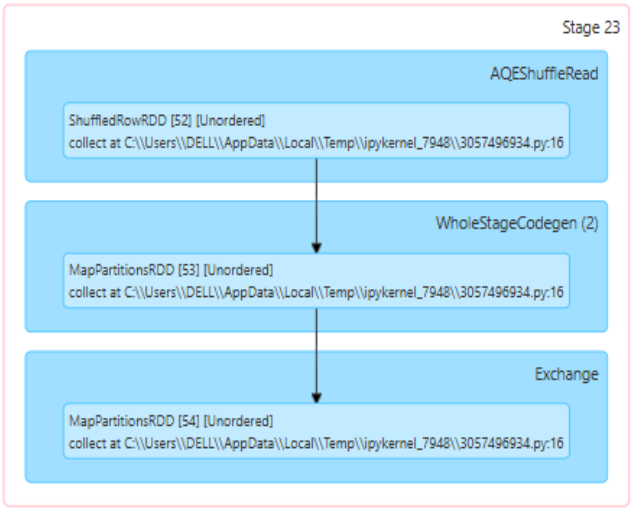
This stage primarily involves narrow dependencies, where each partition works independently on its subset of the data without relying on other partitions.

Optimization Techniques (WholeStageCodegen):

Spark applies WholeStageCodegen to fuse multiple transformations into a single execution plan. This reduces the overhead of intermediate computations, enhancing performance and execution speed.

Final Processing (Exchange):

The stage concludes with an Exchange operation, redistributing data across partitions to enable global operations such as joins, aggregations, or sorting. This introduces wide dependencies, ensuring that the data is correctly organized for the next stage.



Stage 23 Explanation

AQEShuffleRead (ShuffledRowRDD):

The stage begins with an Adaptive Query Execution (AQE) ShuffleRead operation using ShuffledRowRDD. This step reads data shuffled from a previous stage, dynamically adjusting partition sizes for optimized workload balance and efficient processing.

Intermediate Steps (MapPartitionsRDD):

The shuffled data is processed within partitions using a MapPartitionsRDD, ensuring it is structured and optimized for further transformations.

Dependency Type:

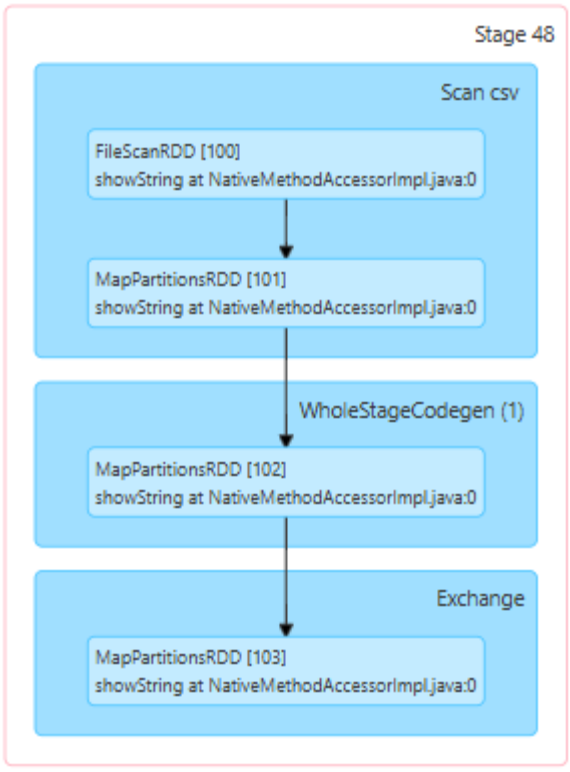
This stage involves wide dependencies, as partitions depend on data from multiple other partitions. This redistribution is necessary for global transformations like joins, sorting, or aggregations.

Optimization Techniques (WholeStageCodegen):

Spark applies WholeStageCodegen to merge multiple transformations into a single execution plan. This reduces intermediate overhead and enhances performance by executing transformations efficiently.

Final Processing (Exchange):

The stage concludes with an Exchange operation, redistributing the data across partitions once more to ensure proper alignment for further global computations in the subsequent stages.



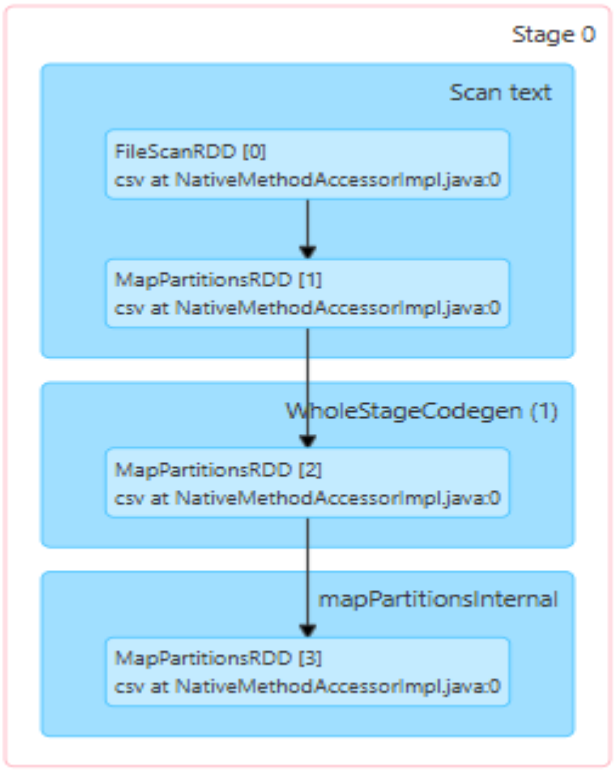
Stage 48 Explanation

Scan CSV (FileScanRDD):
The stage begins with a FileScanRDD operation that reads raw data from a CSV file. This is the initial step to load the dataset into Spark's memory for processing.
Intermediate Steps (MapPartitionsRDD):
The loaded data is processed using a MapPartitionsRDD, ensuring the data is partitioned and structured for efficient execution of transformations in subsequent steps.
Dependency Type:
This stage primarily involves narrow dependencies, where each partition processes its subset of data independently without communication with other partitions.
Optimization Techniques (WholeStageCodegen):
Spark applies WholeStageCodegen to optimize the execution by fusing multiple transformations into a single execution plan. This minimizes overhead from intermediate steps and accelerates processing.
Final Processing (Exchange):
The stage concludes with an Exchange operation, where data is shuffled across partitions to enable global operations such as joins or aggregations in subsequent stages. This introduces wide dependencies, as data is redistributed between partitions to ensure correct processing for downstream transformations.

Stage 50 Explanation

AQEShuffleRead (ShuffledRowRDD):
The stage begins with an Adaptive Query Execution (AQE) ShuffleRead operation using ShuffledRowRDD. This operation dynamically reads shuffled data from earlier stages and optimizes partition sizes to ensure balanced workload and efficient processing.
Intermediate Steps (MapPartitionsRDD):
The shuffled data is processed using a MapPartitionsRDD, which organizes the data within partitions to ensure it is structured for efficient downstream computations.
Dependency Type:
This stage involves wide dependencies, as partitions depend on data from multiple other partitions. This data redistribution is crucial for global transformations like sorting or aggregations.
Optimization Techniques (WholeStageCodegen):
Spark applies WholeStageCodegen to optimize execution by fusing multiple operations into a single execution plan. This minimizes intermediate processing overhead and improves computational performance.
Final Processing (mapPartitionsInternal):
The stage concludes with mapPartitionsInternal, where partition-level computations are performed efficiently. This ensures all required transformations are completed within each partition before moving to the next stage.

STEP 5



Stage 0 Explanation

Scan Text (FileScanRDD):

The stage starts with a FileScanRDD operation, which reads raw text data (e.g., CSV files) from the input source. This is the initial step to load the dataset into memory for processing.

Intermediate Steps (MapPartitionsRDD):

The loaded data is processed using a MapPartitionsRDD. This partitions the data, ensuring it is structured and ready for optimized transformations in subsequent stages.

Dependency Type:

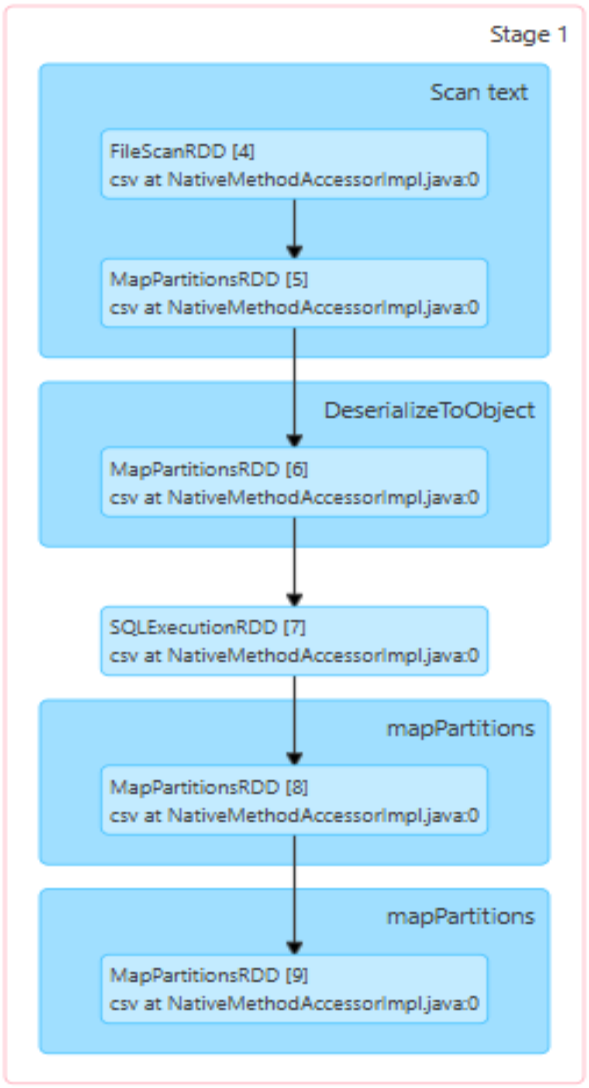
This stage involves narrow dependencies, where each partition processes its subset of data independently without requiring communication between partitions.

Optimization Techniques (WholeStageCodegen):

Spark applies WholeStageCodegen to combine multiple transformations into a single execution plan. This reduces the overhead of intermediate steps and enhances computational performance.

Final Processing (mapPartitionsInternal):

The stage concludes with mapPartitionsInternal, where efficient computations are performed within each partition. This ensures all transformations are completed before transitioning to the next stage.



Stage 1 Explanation

Scan Text (FileScanRDD):

The stage starts with a FileScanRDD operation, reading raw text data (e.g., CSV files) from the input source. This step loads the data into memory for processing.

Intermediate Steps (DeserializeToObject - MapPartitionsRDD):

The raw text data is processed through a MapPartitionsRDD, where it is deserialized into structured objects such as rows or records. This transformation prepares the data for SQL-based or logical computations.

SQL Execution (SQLExecutionRDD):

The deserialized data is passed through SQLExecutionRDD, where SQL queries or DataFrame transformations are executed. This step applies logical operations and computations as defined in the query.

Partition-Level Refinement (MapPartitionsRDD):

Two successive MapPartitionsRDD operations refine the data within each partition, ensuring efficient transformations at a localized level.

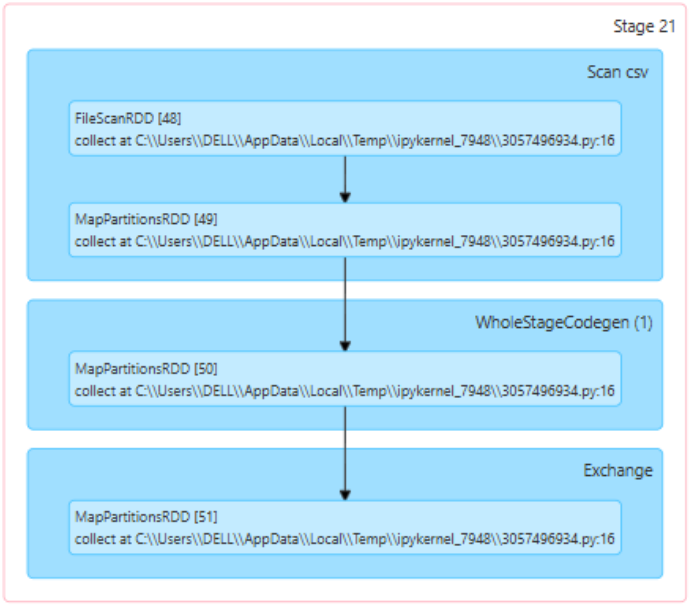
Dependency Type:

This stage primarily involves narrow dependencies, where each partition processes its data independently, avoiding costly inter-partition communication.

Final Processing (MapPartitions):

The stage concludes with final computations in

MapPartitionsRDD, ensuring that all required transformations are completed efficiently within each partition before the next stage begins.



Stage 21 Explanation

Scan CSV (FileScanRDD):

The stage begins with a FileScanRDD operation that reads raw data from a CSV file. This step initializes the process by loading the dataset into memory for further transformations.

Intermediate Steps (MapPartitionsRDD):

The data is processed using a MapPartitionsRDD, partitioning and structuring the data for efficient execution in subsequent steps.

Dependency Type:

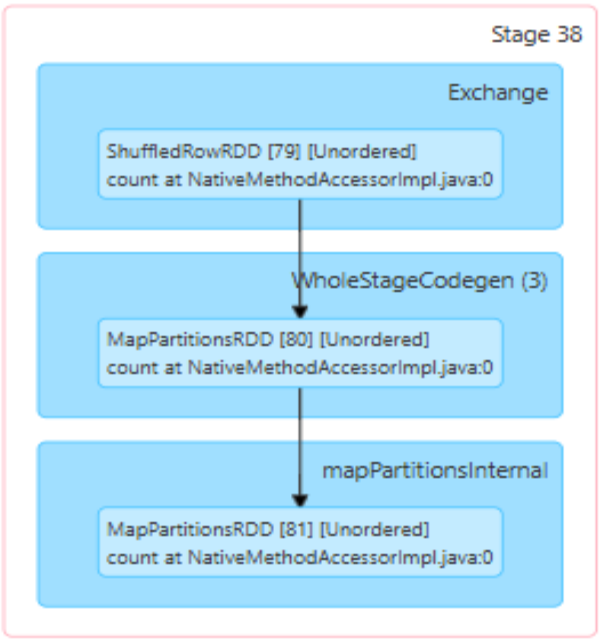
This stage primarily involves narrow dependencies, where each partition processes its subset of data independently, avoiding costly inter-partition communication.

Optimization Techniques (WholeStageCodegen):

Spark applies WholeStageCodegen to fuse multiple operations into a single execution plan. This optimization reduces intermediate processing overhead and accelerates execution speed.

Final Processing (Exchange):

The stage concludes with an Exchange operation, where data is redistributed across partitions to prepare for global transformations such as joins or aggregations. This introduces wide dependencies, ensuring data alignment for downstream operations.



Stage 38 Explanation

Exchange (ShuffledRowRDD):

The stage starts with an Exchange operation using ShuffledRowRDD. This operation redistributes data across partitions, enabling global transformations such as aggregations, joins, or sorts.

Intermediate Steps (MapPartitionsRDD):

The redistributed data is processed by a MapPartitionsRDD. This ensures the data is correctly structured and optimized for subsequent transformations.

Dependency Type:

This stage involves wide dependencies, as each partition relies on data from multiple other partitions. This is necessary for global operations but increases complexity and resource usage.

Optimization Techniques (WholeStageCodegen):

Spark applies WholeStageCodegen to optimize transformations by fusing multiple operations into a single execution plan. This reduces the overhead of intermediate steps and improves computational performance.

Final Processing (mapPartitionsInternal):

The stage concludes with mapPartitionsInternal, where partition-level computations are executed efficiently. This ensures all transformations are completed before transitioning to the next stage.

Stage 50 Explanation

AQEShuffleRead (ShuffledRowRDD):

The stage starts with an Adaptive Query Execution (AQE) ShuffleRead operation using ShuffledRowRDD. This step dynamically reads shuffled data from prior stages, optimizing partition sizes for efficient workload distribution and computation.

Intermediate Steps (MapPartitionsRDD):

A MapPartitionsRDD processes the shuffled data within each partition, ensuring the data is appropriately structured for downstream transformations and computations.

Dependency Type:

This stage involves wide dependencies, as data from each partition depends on multiple other partitions. This redistribution of data is essential for global transformations like joins, aggregations, or sorting.

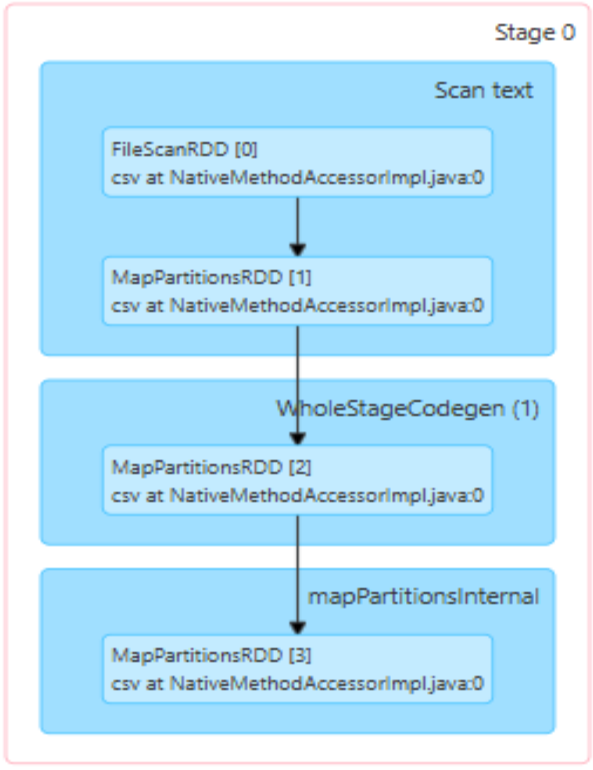
Optimization Techniques (WholeStageCodegen):

Spark applies WholeStageCodegen to optimize the execution plan by combining multiple transformations into a single stage. This reduces intermediate overhead, enhancing performance and execution speed.

Final Processing (mapPartitionsInternal):

The stage concludes with mapPartitionsInternal, where efficient computations are performed at the partition level. This ensures that all required transformations are completed before moving to the next stage.

STEP 6



Stage 0 Explanation

Scan Text (FileScanRDD):

The stage begins with a FileScanRDD operation, which reads raw text data (e.g., CSV files) from the input source. This step initializes the data loading process, bringing the dataset into memory for processing.

Intermediate Steps (MapPartitionsRDD):

The loaded data is processed using a MapPartitionsRDD. This step ensures that the data is divided into partitions and organized for efficient processing in the following stages.

Dependency Type:

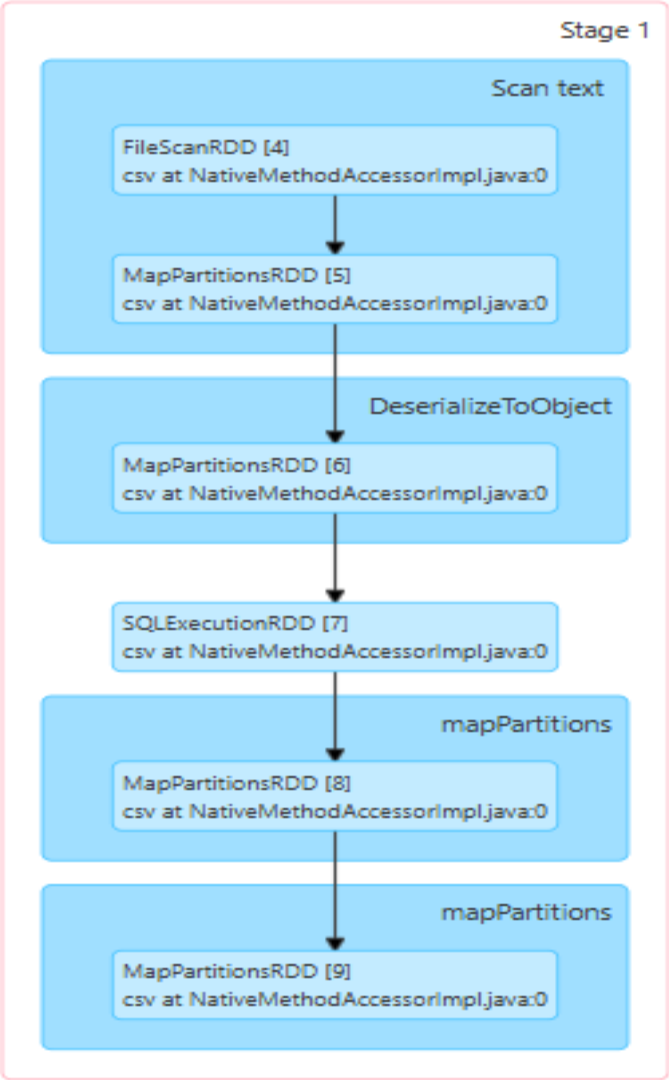
This stage involves narrow dependencies, meaning that each partition operates independently on its own subset of data without requiring communication with other partitions.

Optimization Techniques (WholeStageCodegen):

Spark applies WholeStageCodegen to optimize the execution of transformations. By combining multiple operations into a single execution plan, this step reduces intermediate overhead and accelerates computation.

Final Processing (mapPartitionsInternal):

The stage concludes with mapPartitionsInternal, where efficient computations are carried out at the partition level. This step ensures that all transformations are completed before transitioning to the next stage.



Stage 1 Explanation

Scan Text (FileScanRDD):

The stage begins with a FileScanRDD operation that reads raw text data (e.g., CSV files) from the source. This is the initial step, loading the data into Spark for processing.

Intermediate Steps (DeserializeToObject - MapPartitionsRDD):

The raw text is deserialized into structured objects (e.g., rows) using MapPartitionsRDD. This step organizes the data into a structured format suitable for further transformations and computations.

SQL Execution (SQLExecutionRDD):

The structured data is processed through SQLExecutionRDD, which executes SQL queries or DataFrame transformations to apply logical computations and filters defined in the query.

Partition-Level Transformations (MapPartitionsRDD):

Two MapPartitionsRDD operations refine the data within each partition, applying local transformations efficiently while avoiding data movement between partitions.

Dependency Type:

This stage features narrow dependencies, where each partition processes its own data independently, minimizing overhead and avoiding shuffling.

Final Processing (MapPartitions):

The final MapPartitionsRDD operations perform the necessary transformations and computations, ensuring all tasks are completed before transitioning to the next stage. These steps finalize the data preparation and filtering process.

Stage 21 Explanation

Scan CSV (FileScanRDD):

The stage begins with a FileScanRDD operation, which reads raw data from a CSV file. This is the first step to load the dataset into Spark memory for subsequent transformations.

Intermediate Steps (MapPartitionsRDD):

The scanned data is processed using a MapPartitionsRDD. This partitions and structures the data, ensuring it is ready for efficient computation in subsequent steps.

Dependency Type:

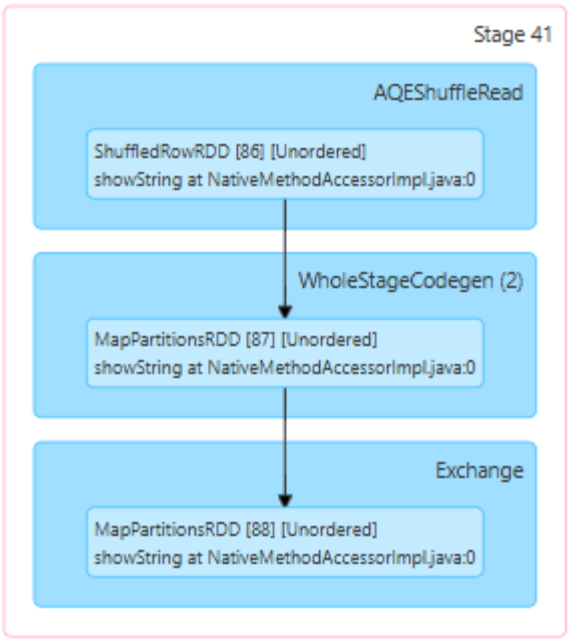
This stage primarily involves narrow dependencies, meaning each partition processes its subset of the data independently, minimizing communication between partitions.

Optimization Techniques (WholeStageCodegen):

Spark applies WholeStageCodegen to optimize execution by fusing multiple transformations into a single execution plan. This step reduces the overhead of intermediate operations, improving efficiency and performance.

Final Processing (Exchange):

The stage concludes with an Exchange operation, where data is redistributed across partitions. This introduces wide dependencies, as the data is reshuffled to enable global operations like joins or aggregations in subsequent stages.



Stage 41 Explanation

AQEShuffleRead (ShuffledRowRDD):

The stage begins with an Adaptive Query Execution (AQE) ShuffleRead operation using ShuffledRowRDD. This step reads shuffled data from a previous stage and dynamically adjusts partition sizes for workload balance and efficiency.

Intermediate Steps (MapPartitionsRDD):

The shuffled data is processed within partitions using a MapPartitionsRDD, ensuring the data is organized and optimized for further transformations.

Dependency Type:

This stage involves wide dependencies, where data in each partition depends on multiple other partitions. This redistribution is essential for global operations like sorting or aggregations.

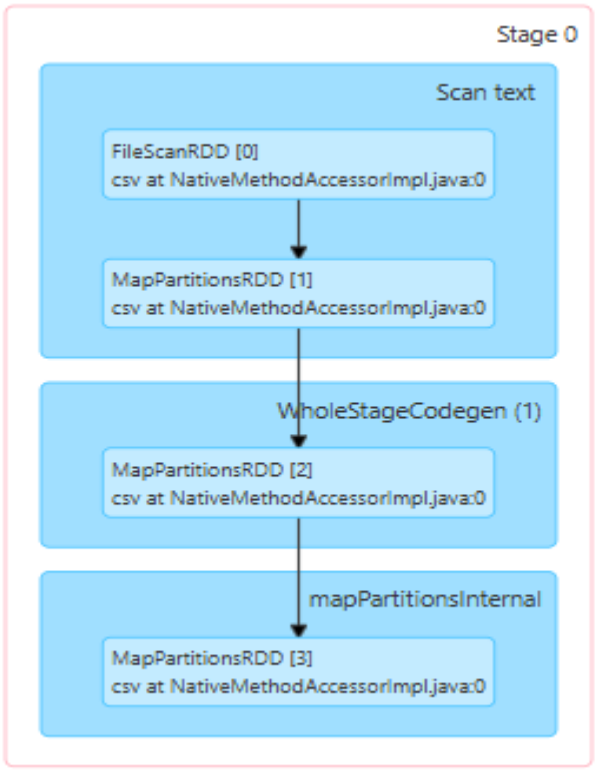
Optimization Techniques (WholeStageCodegen):

Spark applies WholeStageCodegen to fuse multiple transformations into a single execution plan. This reduces intermediate processing overhead and accelerates execution.

Final Processing (Exchange):

The stage concludes with an Exchange operation, where data is reshuffled again across partitions. This ensures proper data alignment for subsequent global computations in the next stages.

STEP 7



Stage 0 Explanation

Scan Text (FileScanRDD):

The stage begins with a FileScanRDD operation that reads raw text data (e.g., CSV files) from the input source. This is the initial step in the data pipeline, loading the dataset into memory for processing.

Intermediate Steps (MapPartitionsRDD):

The data is processed using a MapPartitionsRDD, which partitions the dataset into manageable chunks. This ensures efficient computation in the next stages.

Dependency Type:

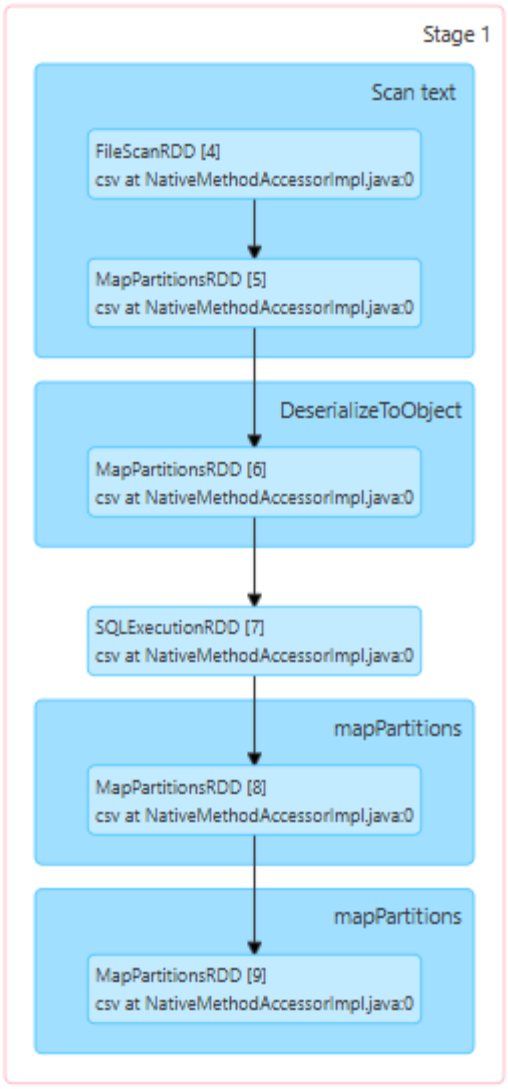
This stage involves narrow dependencies, where each partition operates independently on its subset of the data without communication with other partitions.

Optimization Techniques (WholeStageCodegen):

Spark applies WholeStageCodegen to combine multiple operations into a single execution plan. This optimization minimizes intermediate overhead and accelerates the computation.

Final Processing (mapPartitionsInternal):

The stage concludes with mapPartitionsInternal, where efficient computations are performed within each partition. This ensures all required transformations are completed before transitioning to the next stage.



Stage 1 Explanation

Scan Text (FileScanRDD):

The stage begins with a FileScanRDD operation, which reads raw text data (e.g., CSV files) from the input source. This is the initial step to load the dataset into memory for processing.

Intermediate Steps (DeserializeToObject - MapPartitionsRDD):

The raw text data is processed using a MapPartitionsRDD to deserialize it into structured objects (e.g., rows or records). This step organizes the data into a structured format suitable for logical and SQL-based operations.

SQL Execution (SQLExecutionRDD):

The structured data is passed through SQLExecutionRDD, where SQL transformations or DataFrame operations are applied. This step enables filtering, transformation, or aggregation based on user-defined queries.

Partition-Level Refinement (MapPartitionsRDD):

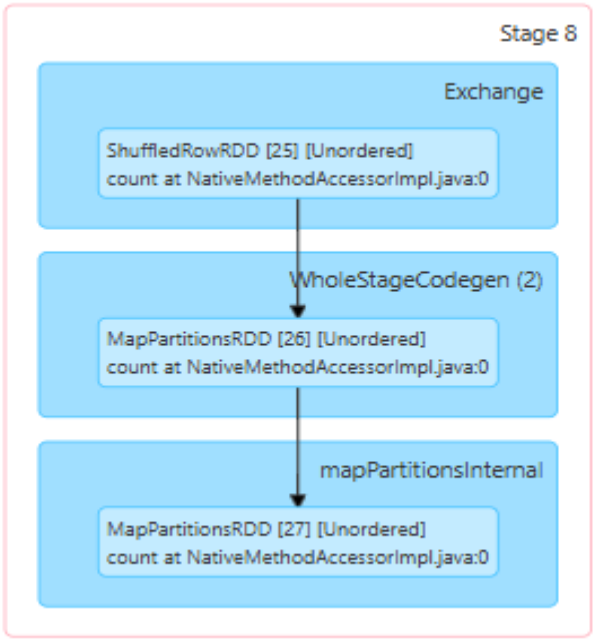
Two additional MapPartitionsRDD steps perform localized computations within partitions. These steps further refine the data and ensure efficient execution without requiring inter-partition communication.

Dependency Type:

This stage involves narrow dependencies, where each partition processes its own subset of data independently, minimizing the overhead of data shuffling.

Final Processing (MapPartitions):

The stage concludes with final MapPartitionsRDD transformations, ensuring that all required computations are complete within each partition before transitioning to the next stage. These steps finalize the partition-level operations efficiently.



Stage 8 Explanation

Exchange (ShuffledRowRDD):

The stage begins with an Exchange operation using ShuffledRowRDD, where data is redistributed across partitions. This shuffling is necessary for global transformations, such as aggregations or joins, to ensure the correct grouping of data.

Intermediate Steps (MapPartitionsRDD):

The redistributed data is processed by a MapPartitionsRDD. This step organizes the shuffled data within each partition for efficient computation in the following steps.

Dependency Type:

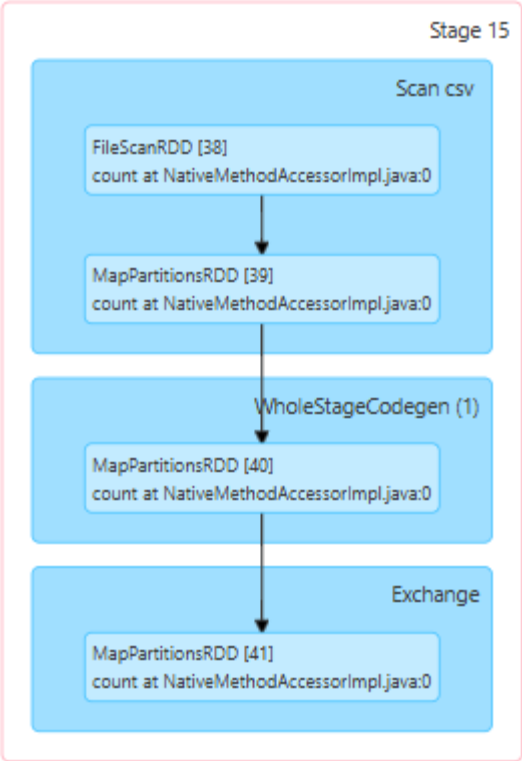
This stage involves wide dependencies, where each partition relies on data from multiple other partitions. This increases computational complexity but is essential for global operations.

Optimization Techniques (WholeStageCodegen):

Spark applies WholeStageCodegen to optimize the execution plan by fusing multiple operations into a single stage. This reduces intermediate overhead and enhances performance.

Final Processing (mapPartitionsInternal):

The stage concludes with mapPartitionsInternal, where efficient partition-level computations are performed. This ensures all required transformations are completed within partitions before transitioning to the next stage.



Stage 15 Explanation

Scan CSV (FileScanRDD):

The stage begins with a FileScanRDD operation, where raw data is read from a CSV file. This is the initial step to load the dataset into memory for further transformations and computations.

Intermediate Steps (MapPartitionsRDD):

The data is processed using a MapPartitionsRDD, ensuring it is structured and partitioned for efficient execution in the following steps.

Dependency Type:

This stage involves narrow dependencies, where each partition operates independently on its subset of the data. This avoids the need for communication between partitions.

Optimization Techniques (WholeStageCodegen):

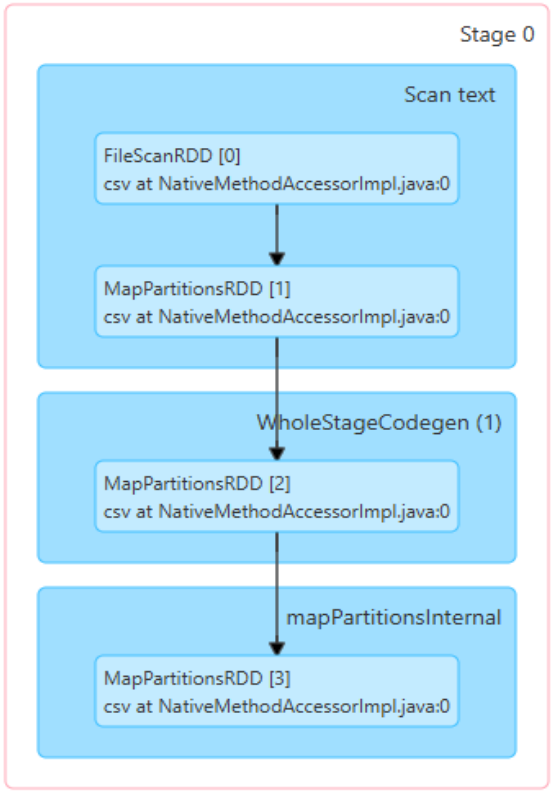
Spark applies WholeStageCodegen to optimize execution by fusing multiple transformations into a single plan. This reduces intermediate overhead and accelerates processing.

Final Processing (Exchange):

The stage concludes with an Exchange operation, where data is redistributed across partitions. This introduces wide dependencies, ensuring the data is properly aligned for global operations, such as joins or aggregations, in subsequent stages.

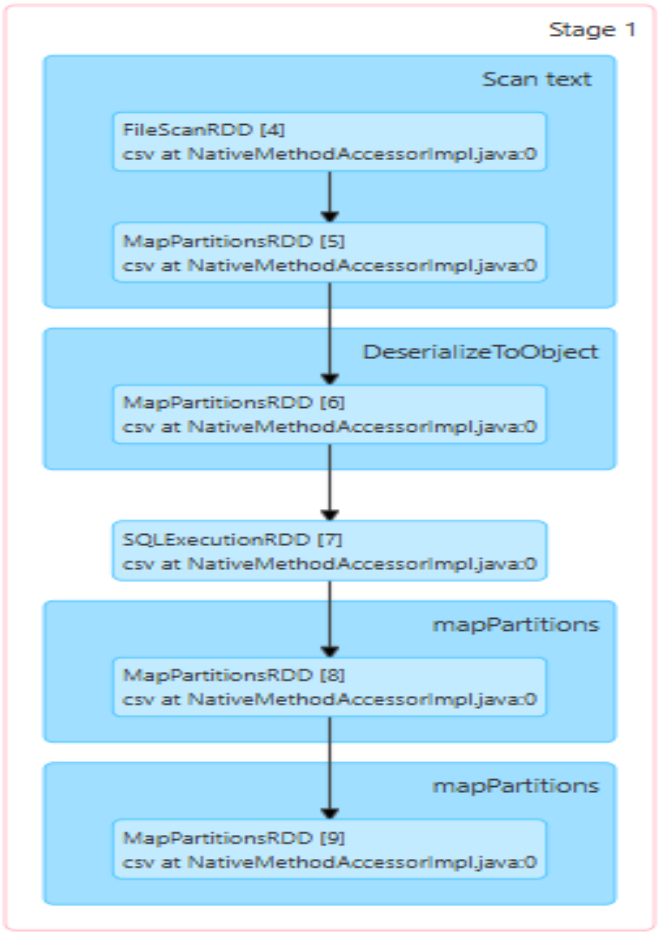
STEP 8

Explanation: Stage 0



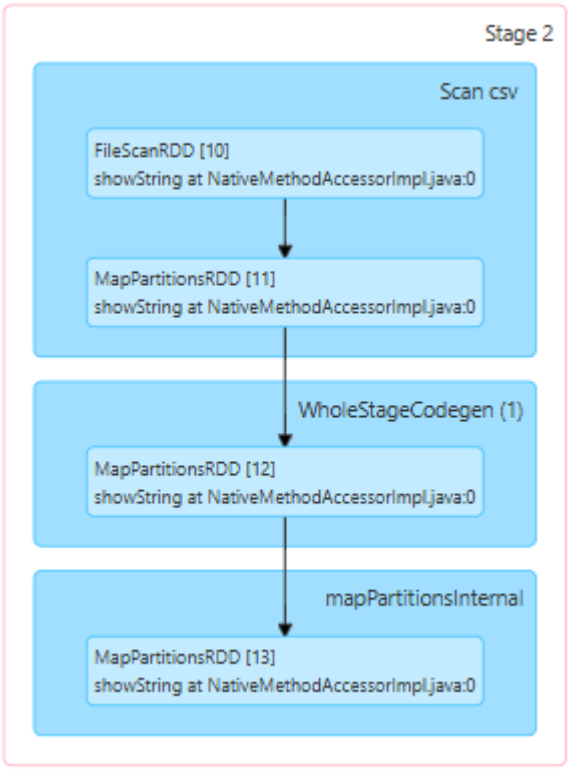
1. Scan Text (FileScanRDD): This stage begins by reading the CSV file using a FileScanRDD. The operation is parallelized across the cluster, allowing efficient data loading into memory from the source file.
2. Intermediate Steps (MapPartitionsRDD): The data is split into multiple partitions and mapped using MapPartitionsRDD. These transformations ensure the data is divided into manageable chunks for processing across the cluster, serving as intermediate steps for further transformations.
3. Optimization Techniques (WholeStageCodegen): Spark applies WholeStageCodegen to optimize this stage by combining several transformations into a single physical plan. This reduces the overhead caused by multiple shuffles and serialization, ensuring better performance.
4. Dependency Type (Narrow Dependency): All transformations in this stage are narrow dependencies, meaning each partition processes its data independently without requiring shuffles or data exchange between partitions. This type of dependency optimizes parallelism and minimizes overhead.
5. Final Processing (mapPartitionsInternal): The final step involves executing mapPartitionsInternal, which is optimized for processing data within each partition. This ensures efficient handling of partition-level tasks before moving to the next stage.

Explanation: Stage 1



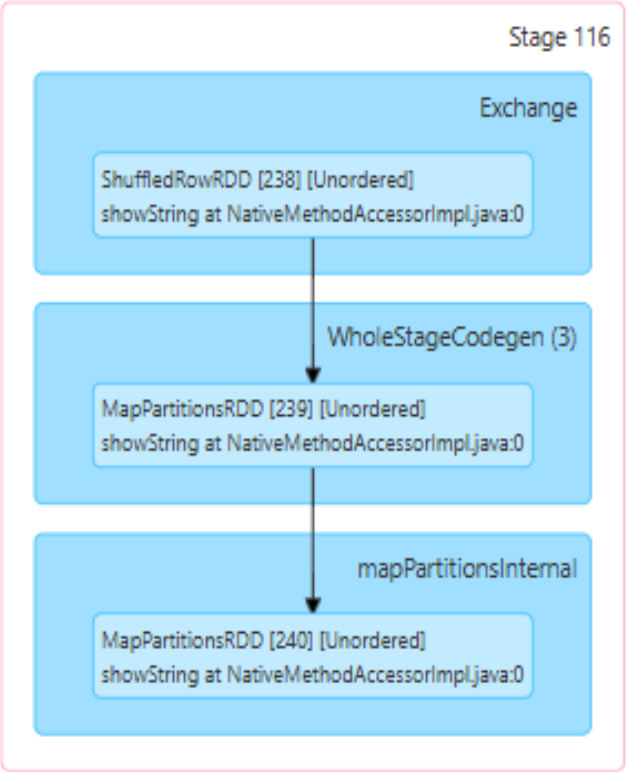
1. Scan Text (FileScanRDD): This stage starts with reading the CSV file as text using FileScanRDD. The data is loaded in parallel across the cluster, allowing for efficient access to the data source.
2. Intermediate Steps (MapPartitionsRDD): After reading the data, it is split into partitions using MapPartitionsRDD to facilitate parallel processing. These steps divide the dataset into chunks, preparing it for further transformations.
3. DeserializeToObject: In this step, the data is deserialized into objects. This process transforms the raw data into a structured format, enabling Spark to interpret and process the data efficiently in subsequent steps.
4. SQL Execution (SQLExecutionRDD): Here, Spark leverages SQL execution capabilities, applying transformations and actions through SQLExecutionRDD. This step integrates query execution logic to optimize operations on structured data.
5. mapPartitions (MapPartitionsRDD): The data undergoes additional mapPartitions transformations. Each partition is processed independently, with Spark applying the defined logic to partitioned data.
6. Dependency Type (Narrow Dependency): Like Stage 0, all operations in this stage involve narrow dependencies. Each partition operates independently, avoiding costly shuffles and ensuring high parallelism.
7. Final Processing (mapPartitions): The final step involves partition-level execution using mapPartitions, completing the transformations on each partition efficiently before transitioning to the next stage.

Explanation: Stage 2



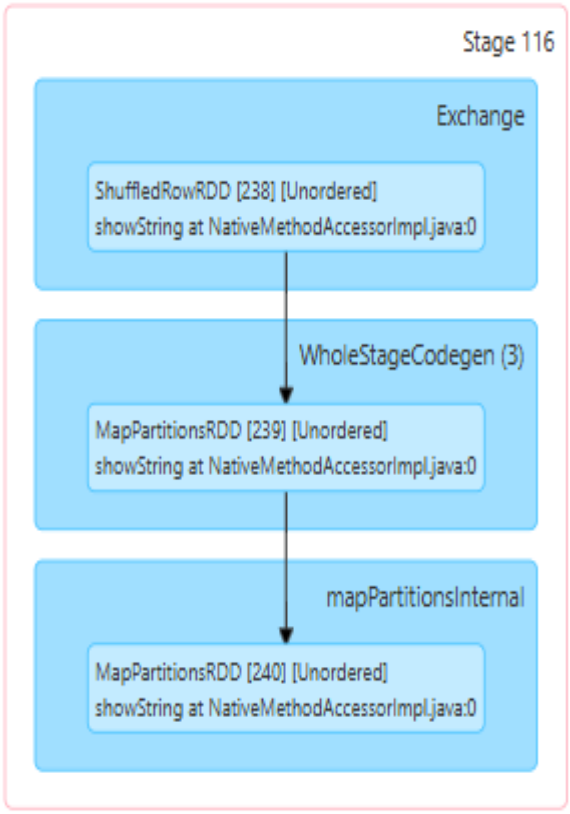
1. Scan CSV (FileScanRDD): The stage begins by scanning the CSV file using FileScanRDD. This step reads the data in parallel across the cluster to ensure efficient access to the input file.
2. Intermediate Steps (MapPartitionsRDD): After scanning, the data is split into partitions using MapPartitionsRDD. Each partition represents a chunk of the dataset, which will undergo further transformations.
3. Optimization Techniques (WholeStageCodegen): Spark applies WholeStageCodegen to optimize the pipeline by combining multiple transformations into a single execution plan. This reduces execution overhead by minimizing data shuffles and improving processing speed.
4. Dependency Type (Narrow Dependency): The transformations in this stage involve narrow dependencies. Each partition operates independently, avoiding the need for shuffles or data exchange between partitions. This ensures high parallelism and efficiency.
5. Final Processing (mapPartitionsInternal): The last step involves processing each partition using mapPartitionsInternal. This step executes optimized operations on each partition, completing the data transformations for this stage.

Explanation: Stage 116



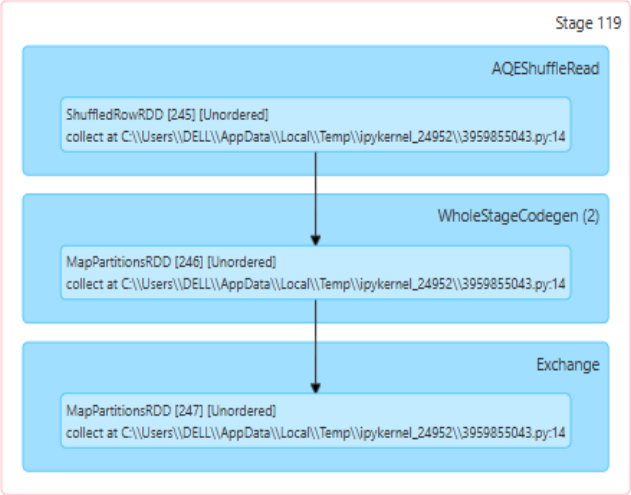
1. Exchange (ShuffledRowRDD): This stage starts with a shuffle operation using ShuffledRowRDD. This step reorganizes the data across partitions based on the key, enabling operations like joins or aggregations that require data redistribution. Shuffling introduces wide dependencies, as data is exchanged between partitions.
2. Intermediate Steps (MapPartitionsRDD): After the shuffle, the data is split into new partitions using MapPartitionsRDD. This transformation prepares the data for further computations in subsequent steps.
3. Optimization Techniques (WholeStageCodegen): Spark applies WholeStageCodegen to optimize the execution plan. By fusing multiple operations into a single physical execution step, Spark minimizes the overhead from intermediate operations, improving the performance of the stage.
4. Dependency Type (Wide Dependency): The shuffle operation introduces wide dependencies, as data from multiple partitions is required to compute results in each partition. This increases the complexity and cost of the operation compared to narrow dependencies.
5. Final Processing (mapPartitionsInternal): The stage concludes with mapPartitionsInternal, which performs the final computations on each partition. This step ensures that partition-level tasks are executed efficiently, completing the transformations for this stage.

Explanation: Stage 116



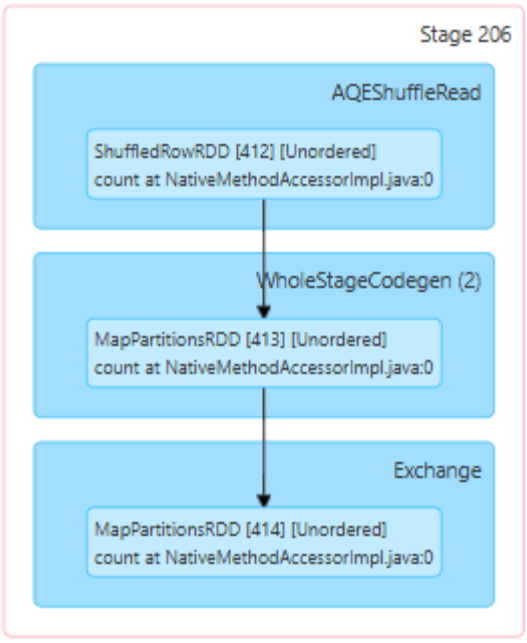
1. Exchange (ShuffledRowRDD): This stage begins with a shuffle operation using ShuffledRowRDD. The shuffle re-partitions the data across the cluster based on keys or criteria needed for subsequent computations like joins or aggregations. Shuffling introduces wide dependencies, meaning data from multiple partitions is exchanged.
2. Intermediate Steps (MapPartitionsRDD): Following the shuffle, MapPartitionsRDD is used to split and process the reorganized data into new partitions. Each partition processes its chunk of data for further computations.
3. Optimization Techniques (WholeStageCodegen): Spark employs WholeStageCodegen to optimize execution. This technique consolidates multiple transformations into a single stage of execution, reducing overhead from intermediate steps and improving overall performance.
4. Dependency Type (Wide Dependency): The shuffle introduces wide dependencies, where data movement across partitions increases complexity and resource usage. These dependencies are necessary for operations requiring global data reorganization.
5. Final Processing (mapPartitionsInternal): The final step involves mapPartitionsInternal, which efficiently executes the partition-level tasks. This step completes the transformations and prepares the data for further stages or output.

Explanation: Stage 119



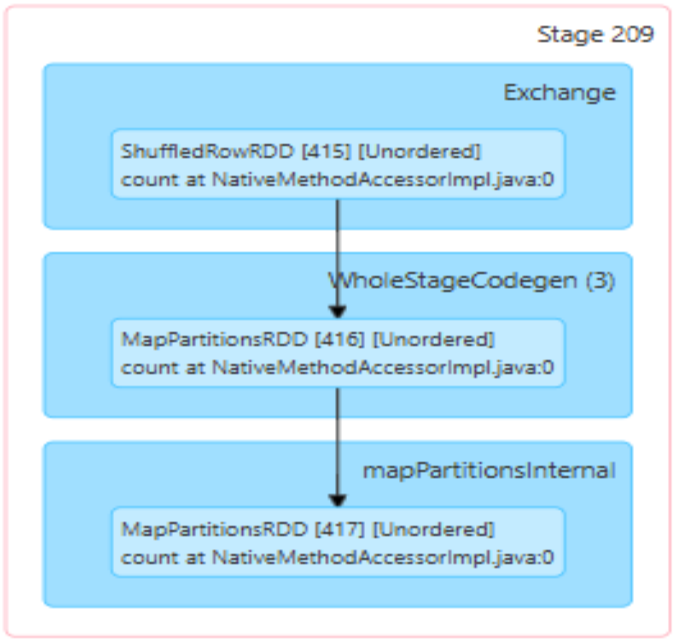
1. AQEShuffleRead (ShuffledRowRDD): This stage begins with AQEShuffleRead, which is an Adaptive Query Execution (AQE) feature that optimizes shuffle read operations. It adjusts the shuffle partitions dynamically based on runtime statistics, ensuring better resource utilization and reducing execution time. This operation reads data from the shuffle output using ShuffledRowRDD.
2. Intermediate Steps (MapPartitionsRDD): The data is processed using MapPartitionsRDD, splitting the shuffled data into partitions for further transformations. Each partition processes its assigned chunk of data independently.
3. Optimization Techniques (WholeStageCodegen): Spark applies WholeStageCodegen to consolidate multiple transformations into a single execution step. This optimization improves performance by minimizing overhead from intermediate steps and reducing the need for additional shuffles or serialization.
4. Dependency Type (Wide Dependency): The shuffle operation introduces wide dependencies as data needs to be exchanged across partitions. However, the AQE optimization reduces the overhead typically associated with such dependencies by dynamically adjusting partition sizes.
5. Final Processing (Exchange): The final step in this stage involves another Exchange operation, likely preparing the data for further computations or ensuring it is in the required format for subsequent stages. This operation re-partitions or redistributes data based on the processing requirements of the next stage.

Explanation: Stage 206



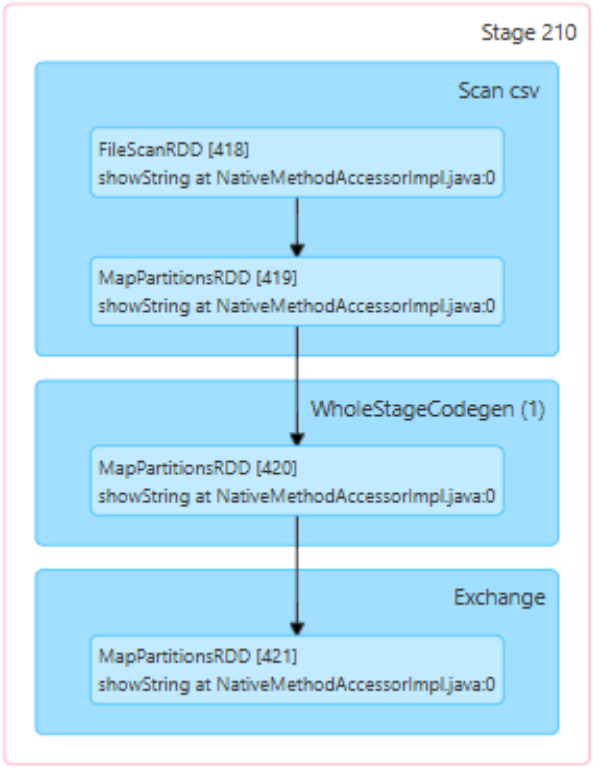
1. AQEShuffleRead (ShuffledRowRDD): This stage starts with an AQEShuffleRead operation using ShuffledRowRDD. Adaptive Query Execution (AQE) dynamically optimizes the shuffle read process by adjusting shuffle partitions at runtime. This ensures efficient resource utilization and minimizes execution time for wide dependency operations.
2. Intermediate Steps (MapPartitionsRDD): The shuffled data is processed and partitioned using MapPartitionsRDD. This transformation splits the data into manageable chunks, allowing parallel processing across the cluster.
3. Optimization Techniques (WholeStageCodegen): Spark leverages WholeStageCodegen to fuse multiple transformations into a single execution plan. This optimization reduces overhead from intermediate operations and improves overall execution efficiency.
4. Dependency Type (Wide Dependency): The shuffle operation introduces wide dependencies, where partitions depend on data from multiple other partitions. AQE optimizations, however, reduce the cost of these dependencies by dynamically managing partition sizes.
5. Final Processing (Exchange): The stage concludes with an Exchange operation, redistributing or re-partitioning the data as needed for subsequent computations. This ensures the data is in the required structure or format for the next stages of processing.

Explanation: Stage 209



1. Exchange (ShuffledRowRDD): This stage starts with an Exchange operation, implemented using ShuffledRowRDD. It involves shuffling data across partitions to enable computations like aggregations or joins that require reorganization of the dataset. This step introduces wide dependencies, where data from multiple partitions is exchanged.
2. Intermediate Steps (MapPartitionsRDD): After the shuffle, MapPartitionsRDD processes the shuffled data into partitions. Each partition operates independently, preparing the data for further transformations or computations.
3. Optimization Techniques (WholeStageCodegen): Spark uses WholeStageCodegen to optimize this stage by fusing multiple operations into a single execution plan. This reduces the overhead of intermediate steps and serialization, improving performance and execution speed.
4. Dependency Type (Wide Dependency): The Exchange operation represents a wide dependency, where each partition relies on data from multiple other partitions. This is essential for operations that require global data reorganization.
5. Final Processing (mapPartitionsInternal): The final step involves mapPartitionsInternal, which executes the partition-level tasks efficiently. This operation ensures the data is fully processed and prepared for subsequent stages or final output.

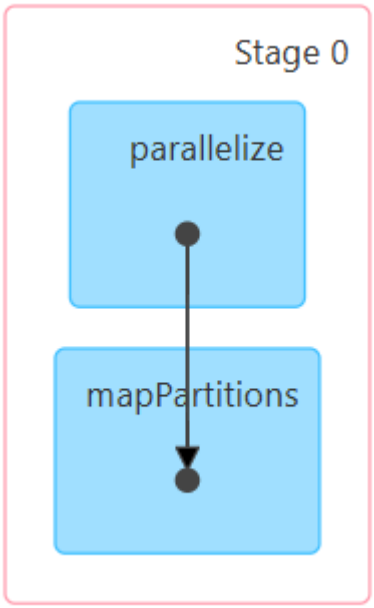
Explanation: Stage 210



1. Scan CSV (FileScanRDD): The stage begins with scanning the CSV file using FileScanRDD. This operation reads the data in parallel across the cluster, efficiently loading it into Spark for further processing.
2. Intermediate Steps (MapPartitionsRDD): After reading the data, it is partitioned into manageable chunks using MapPartitionsRDD. Each partition processes its respective subset of data, preparing it for downstream transformations.
3. Optimization Techniques (WholeStageCodegen): Spark applies WholeStageCodegen to optimize execution by fusing multiple transformations into a single physical plan. This reduces the overhead of intermediate operations and serialization, improving the efficiency of the stage.
4. Dependency Type (Narrow Dependency): The transformations involve narrow dependencies, where each partition processes data independently without requiring shuffles or data exchange across partitions. This ensures a high degree of parallelism and minimizes overhead.
5. Final Processing (Exchange): The stage concludes with an Exchange operation, redistributing or re-partitioning the data as required for subsequent stages. This ensures the data is in the correct structure for further computations or output.

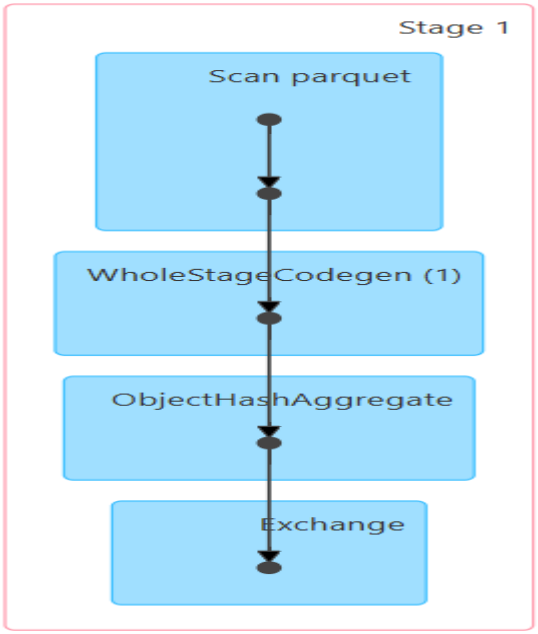
B. Model Training DAG Analysis

Linear regression



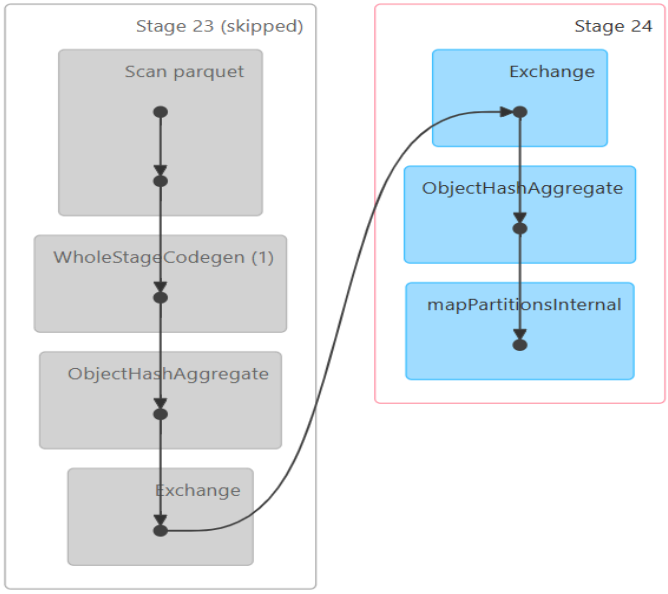
Stage 0 Explanation (Parallelize and mapPartitions)

Parallelize:
The stage begins by creating an RDD through the parallelize operation. This initializes an in-memory collection as an RDD, which partitions the data for parallel processing.
Intermediate Steps (mapPartitions):
The data is processed using the mapPartitions transformation, applying a function to each partition of the RDD independently. This ensures efficient local computation within partitions.
Dependency Type:
This stage involves narrow dependencies, where each partition processes only its local data without requiring input from other partitions.



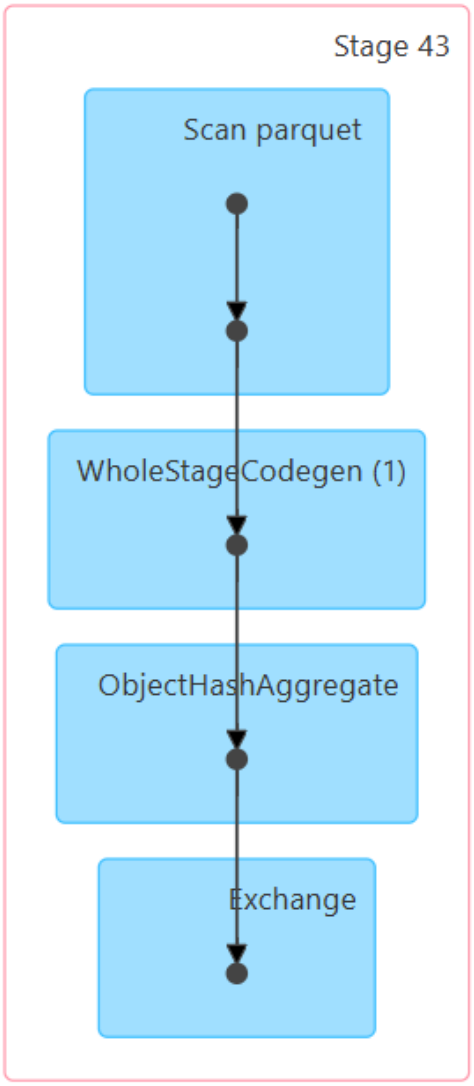
Stage 1 Explanation (Scan parquet and ObjectHashAggregate)

Scan Parquet:
This stage starts by scanning a Parquet file using Scan parquet, reading structured data efficiently in a columnar format.
Optimization Techniques (WholeStageCodegen):
Spark applies WholeStageCodegen to optimize transformations by combining multiple operations into a single execution plan, reducing intermediate steps.
Intermediate Steps (ObjectHashAggregate):
An ObjectHashAggregate operation performs in-memory aggregations using a hash map. This is typically used for efficient grouped operations like sum, average, or count.
Final Processing (Exchange):
The stage concludes with an Exchange operation, where data is shuffled across partitions to enable global transformations like joins or aggregations. This introduces wide dependencies.



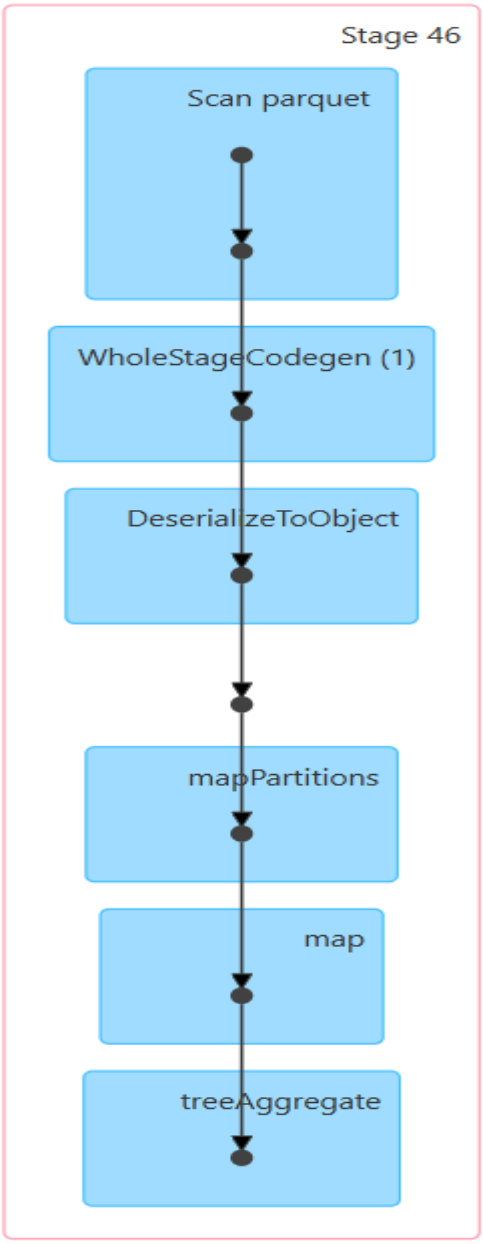
Stage 23 and Stage 24 Explanation (Skipped Stage and Exchange)

Skipped Stage (Stage 23):
This stage is skipped due to Spark optimizations, as redundant computations or transformations are removed during query execution.
Stage 24 Explanation:
Exchange: Redistributes data across partitions to ensure proper alignment for global transformations.
ObjectHashAggregate: Performs aggregation on shuffled data.
mapPartitionsInternal: Finalizes partition-level computations for efficient execution before transitioning to the next stage.



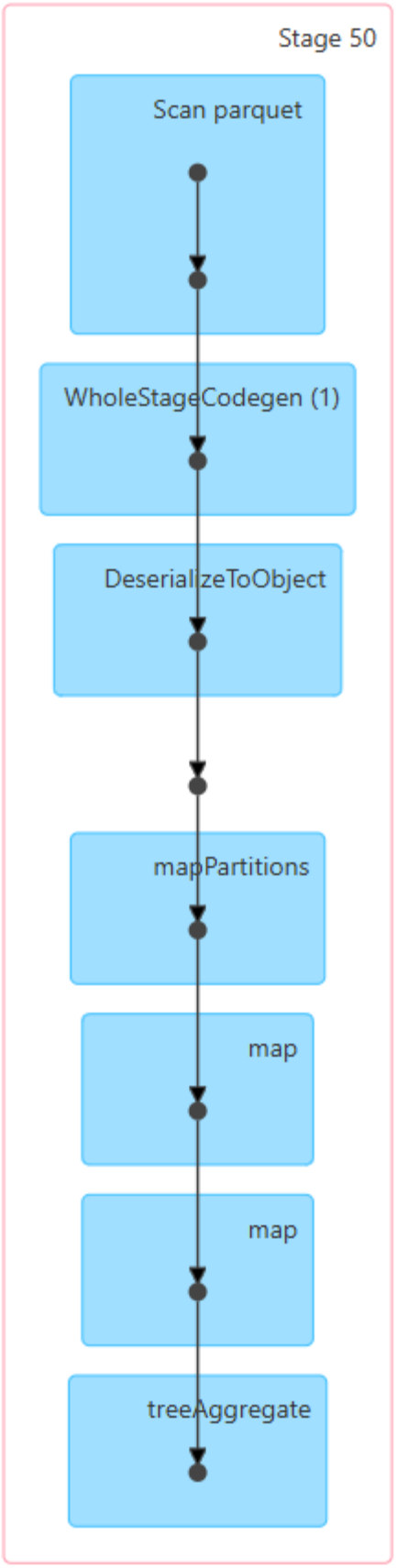
Stage 43 Explanation (Scan parquet and ObjectHashAggregate)

Scan Parquet:
The stage begins with a Scan parquet operation, reading the structured data from a Parquet file.
Optimization Techniques (WholeStageCodegen):
Spark applies WholeStageCodegen to optimize execution, merging transformations into a single execution plan.
Intermediate Steps (ObjectHashAggregate):
Performs aggregations using an in-memory hash map to compute grouped results efficiently.
Final Processing (Exchange):
Redistributes data across partitions for further global operations, introducing wide dependencies.



Stage 46 Explanation (treeAggregate and map transformations)

Scan Parquet:
Starts with reading the Parquet file data for processing.
Optimization Techniques (WholeStageCodegen):
Combines multiple transformations to optimize execution and reduce overhead.
Intermediate Steps (DeserializeToObject, mapPartitions, map):
DeserializeToObject: Converts Parquet data into structured objects (rows).
mapPartitions: Applies a function to each partition independently.
map: Performs transformations on each element in the RDD.
Final Processing (treeAggregate):
Performs an aggregation in a hierarchical manner for efficiency, typically used for reducing or summarizing data.

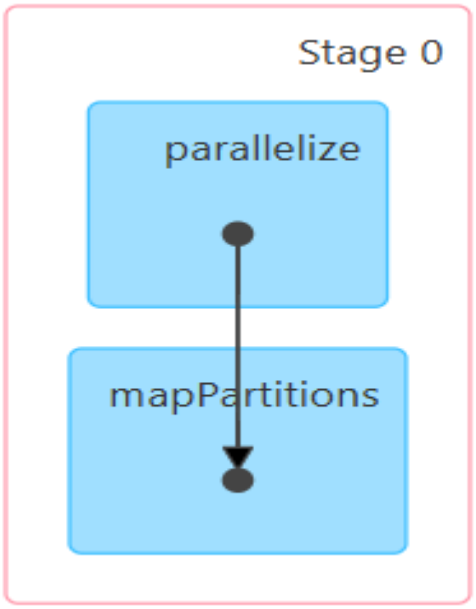


Stage 50 Explanation (treeAggregate and map transformations)

Scan Parquet:
Reads the Parquet file as the first step in the computation pipeline.

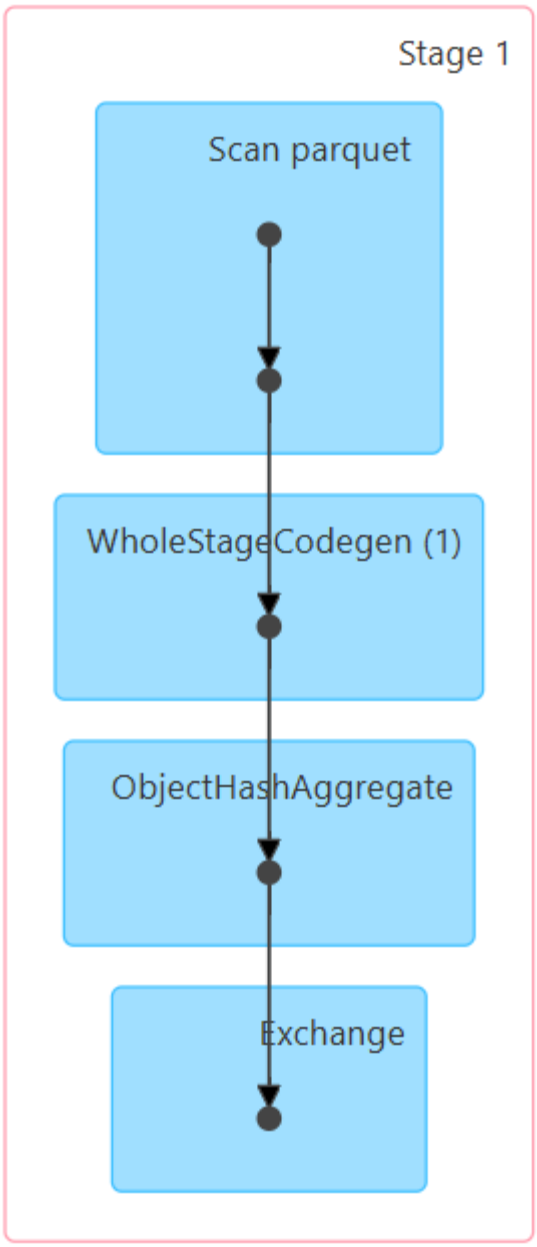
Optimization Techniques (WholeStageCodegen):
Spark optimizes the query by combining multiple operations into one execution plan.
Intermediate Steps (DeserializeToObject, mapPartitions, map):
DeserializeToObject: Converts Parquet data into structured objects.
mapPartitions: Processes partitions independently.
map: Transforms each element in the partition.
Final Processing (treeAggregate):
Aggregates data hierarchically, ensuring efficient and accurate computation for large datasets.

LASSO Regression



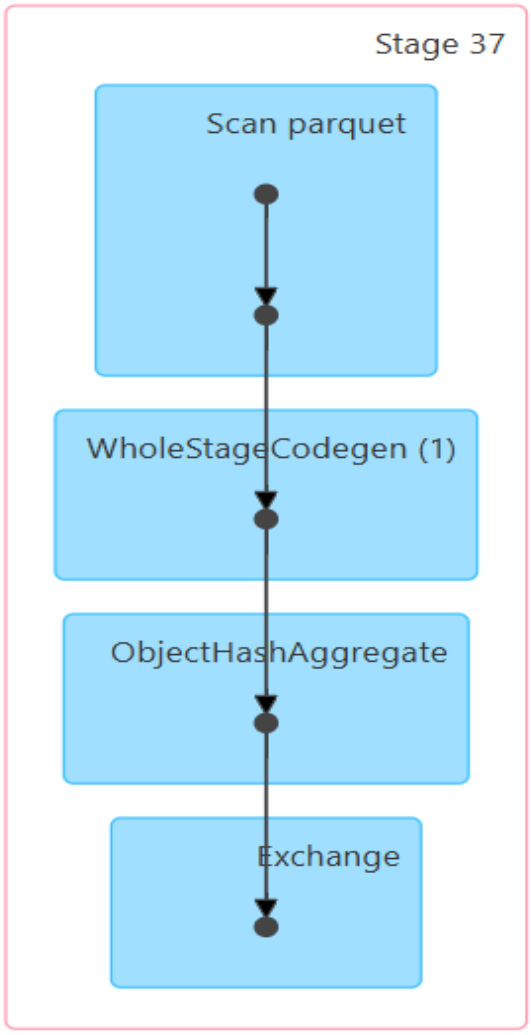
Stage 0 Explanation (Parallelize and mapPartitions)

Parallelize:
The stage begins with the parallelize operation, creating an RDD from an in-memory collection. This step initializes the data structure for distributed processing.
Intermediate Steps (mapPartitions):
The mapPartitions operation applies a transformation to each partition independently, ensuring efficient localized computation without shuffling.
Dependency Type:
This stage has narrow dependencies, meaning each partition processes its own subset of data without interaction with other partitions.



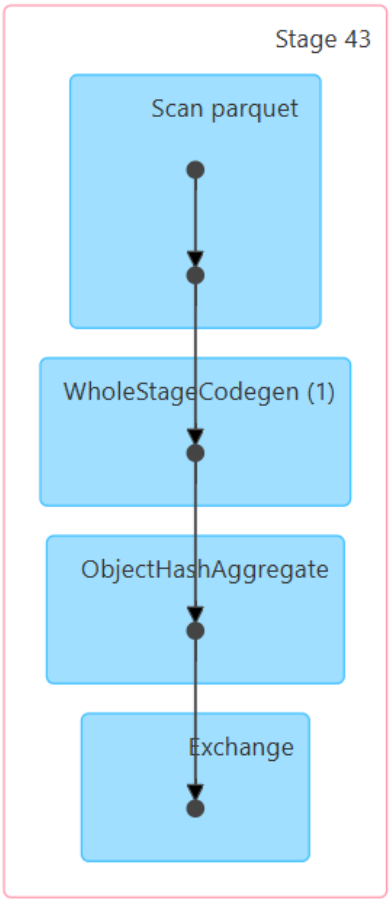
Stage 1 Explanation (Scan Parquet and ObjectHashAggregate)

Scan Parquet:
The stage starts by scanning a Parquet file using Scan parquet. This operation efficiently reads columnar data from the file.
Optimization Techniques (WholeStageCodegen):
Spark applies WholeStageCodegen to fuse multiple transformations into a single optimized execution plan, reducing the overhead of intermediate operations.
Intermediate Steps (ObjectHashAggregate):
An ObjectHashAggregate operation is used to perform grouped aggregations efficiently using in-memory data structures.
Final Processing (Exchange):
Data is redistributed across partitions through an Exchange operation. This introduces wide dependencies as data is shuffled to enable global operations.



Stage 37 Explanation (Similar to Stage 1)

This stage mirrors Stage 1, with the same steps:
Scan Parquet: Reads structured data from a Parquet file.
WholeStageCodegen: Optimizes execution by fusing transformations.
ObjectHashAggregate: Performs aggregation using in-memory structures.
Exchange: Redistributes data for global transformations.



Stage 0 Explanation (Parallelize and mapPartitions)

Parallelize:
The stage begins with the parallelize operation, creating an RDD from an in-memory collection. This step initializes the data structure for distributed processing.
Intermediate Steps (mapPartitions):
The mapPartitions operation applies a transformation to each partition independently, ensuring efficient localized computation without shuffling.
Dependency Type:
This stage has narrow dependencies, meaning each partition processes its own subset of data without interaction with other partitions.

Stage 1 Explanation (Scan Parquet and ObjectHashAggregate)

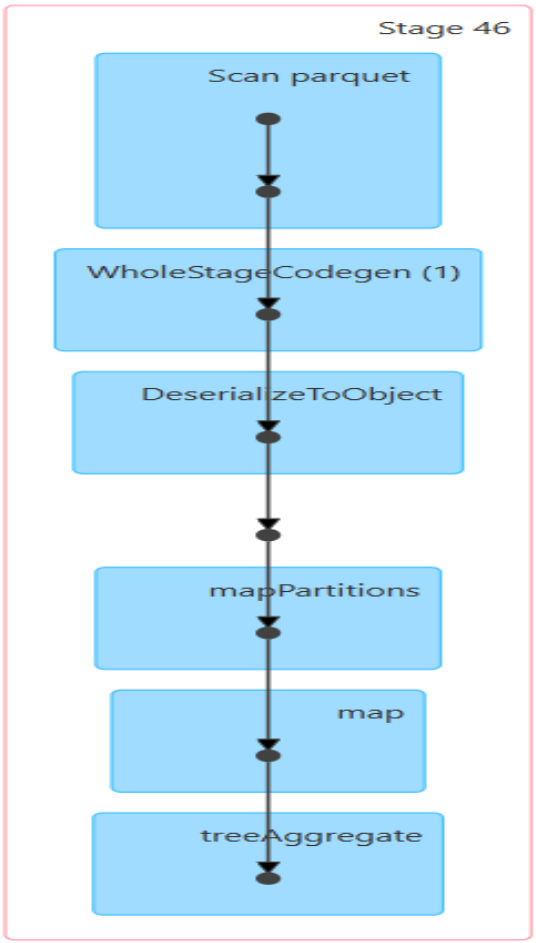
Scan Parquet:
The stage starts by scanning a Parquet file using Scan parquet. This operation efficiently reads columnar data from the file.
Optimization Techniques (WholeStageCodegen):
Spark applies WholeStageCodegen to fuse multiple transformations into a single optimized execution plan, reducing the overhead of intermediate operations.
Intermediate Steps (ObjectHashAggregate):
An ObjectHashAggregate operation is used to perform grouped aggregations efficiently using in-memory data structures.
Final Processing (Exchange):
Data is redistributed across partitions through an Exchange operation. This introduces wide dependencies as data is shuffled to enable global operations.

Stage 37 Explanation (Similar to Stage 1)

This stage mirrors Stage 1, with the same steps:
Scan Parquet: Reads structured data from a Parquet file.
WholeStageCodegen: Optimizes execution by fusing transformations.
ObjectHashAggregate: Performs aggregation using in-memory structures.
Exchange: Redistributes data for global transformations.

Stage 43 Explanation (Scan Parquet and ObjectHashAggregate)

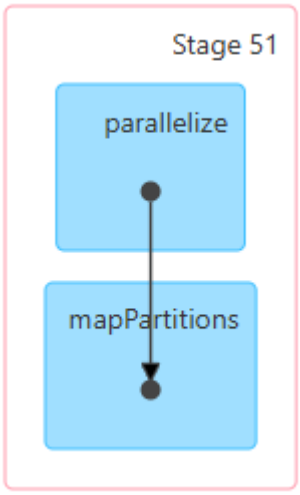
This stage also mirrors Stage 1, repeating the steps:
Scan Parquet: Reads structured data from Parquet.
WholeStageCodegen: Optimizes query execution.
ObjectHashAggregate: Performs efficient in-memory aggregation.
Exchange: Shuffles data for further processing.



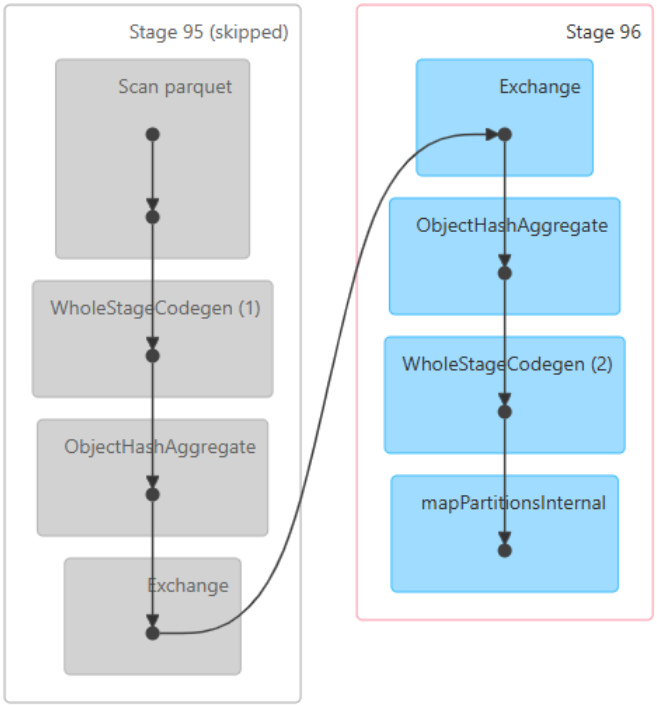
Stage 46 Explanation (treeAggregate and map transformations)

Scan Parquet:
Data is loaded from a Parquet file.
Optimization Techniques (WholeStageCodegen):
Fuses multiple operations for a more efficient execution plan.
Intermediate Steps (DeserializeToObject, mapPartitions,

map):
DeserializeToObject: Converts Parquet data into structured rows.
mapPartitions: Processes partitions independently.
map: Applies transformations to each element.
Final Processing (treeAggregate):
The data is aggregated hierarchically using the treeAggregate operation, ensuring efficient computation of global summaries.

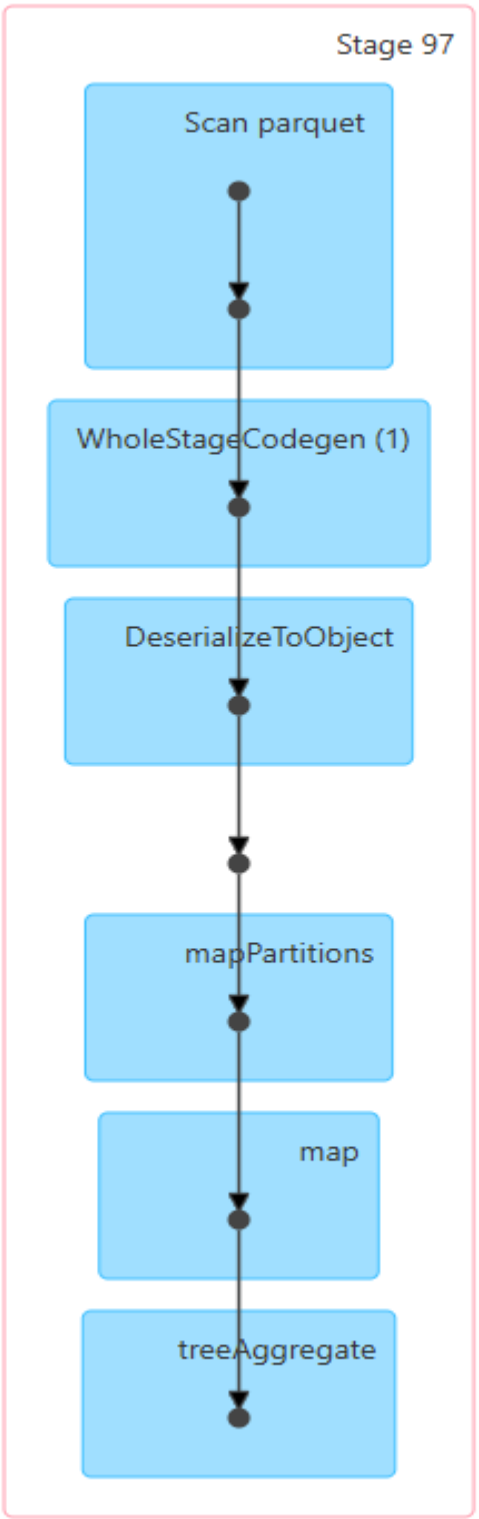


Stage 51 Explanation (Parallelize and mapPartitions)
Parallelize:
Creates an RDD from in-memory data.
mapPartitions:
Processes each partition independently, applying a transformation.
Dependency Type:
This stage has narrow dependencies with localized computation.



Stage 96 Explanation (ObjectHashAggregate and Exchange)

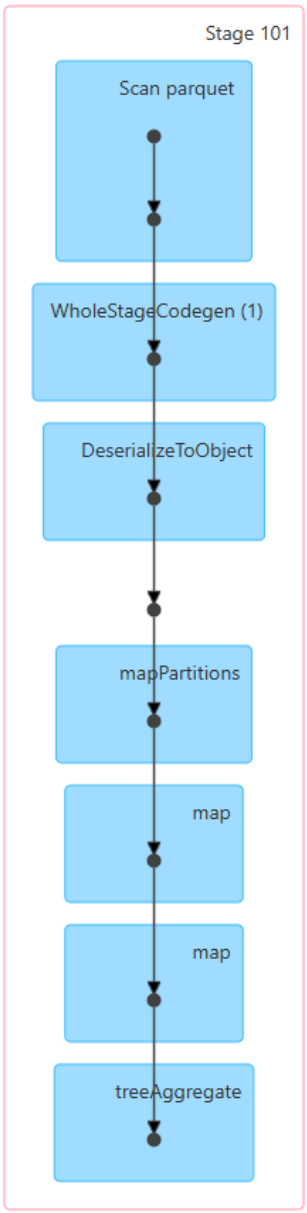
Exchange:
Shuffles data across partitions to align it for aggregation.
ObjectHashAggregate:
Performs grouped aggregations using in-memory data structures.
Optimization Techniques (WholeStageCodegen):
Further optimizes the processing of operations.
Final Step (mapPartitionsInternal):
Completes partition-level computations.



Stage 97 Explanation (treeAggregate and map transformations)

Similar to Stage 46, with:
Scan Parquet: Reads structured data.
WholeStageCodegen: Fuses operations.

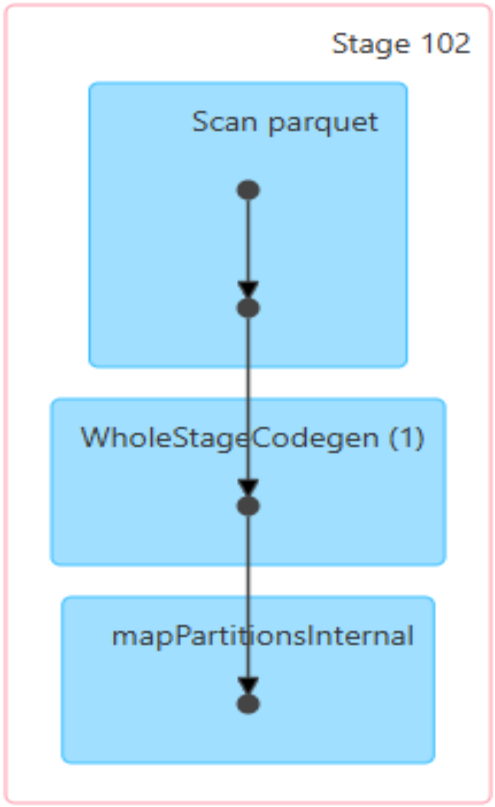
DeserializeToObject, mapPartitions, map: Processes data in structured rows and applies transformations.
treeAggregate: Performs hierarchical aggregation.



Stage 101 Explanation (treeAggregate and map transformations)

Similar to Stage 97, this stage repeats:

Scan Parquet: Loads columnar data.
WholeStageCodegen: Optimizes the plan.
DeserializeToObject, mapPartitions, map: Processes and transforms data.
treeAggregate: Efficiently computes global summaries.

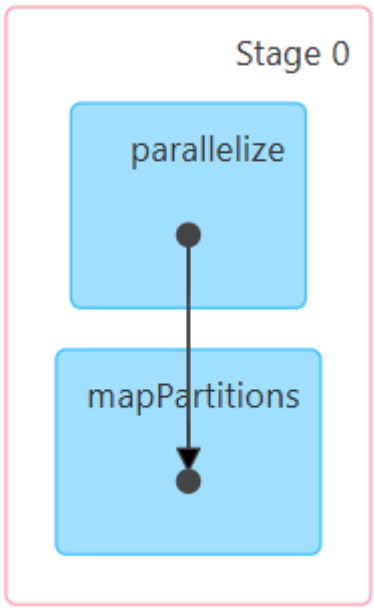


Stage 102 Explanation (Final Scan and Aggregation)

Scan Parquet:
Reads structured data from the file.
Optimization Techniques (WholeStageCodegen):
Combines multiple operations into one optimized execution plan.
mapPartitionsInternal:
Completes the final transformation and partition-level computation, preparing the results for output or subsequent stages.

ELASTIC NET REGRESSION

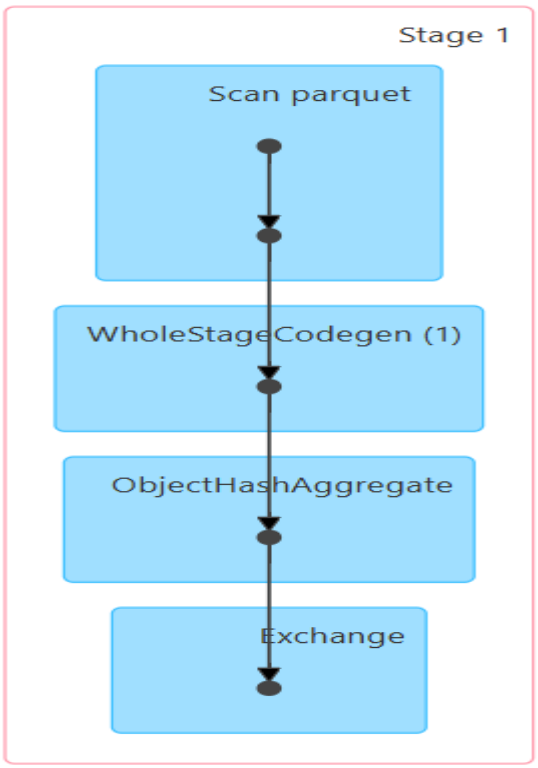
Stage 0: Parallelize



Operations:

1. Parallelize: This stage begins with the parallelization of data, dividing it into smaller chunks for distributed computation.
2. mapPartitions: A transformation is applied to each partition to process data independently.

Dependencies: This stage has narrow dependencies as each partition operates independently without data exchange between partitions.

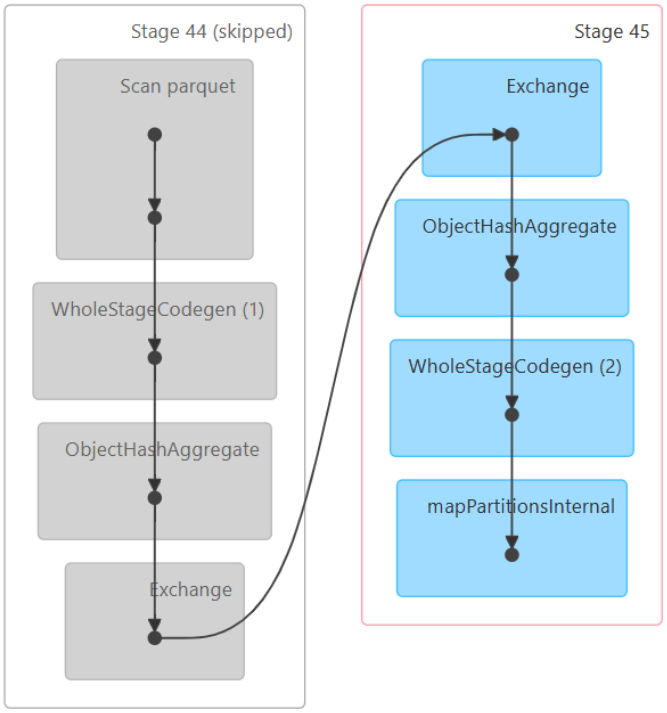


Stage 1: Scan Parquet and Aggregation

Operations:

1. Scan Parquet: Reads Parquet data for processing.
2. WholeStageCodegen (1): Optimizes transformations by fusing multiple operations into a single physical execution plan.
3. ObjectHashAggregate: Performs aggregation using in-memory hashing.
4. Exchange: A shuffle operation occurs, redistributing data across partitions for downstream processing.

Dependencies: This stage includes wide dependencies due to the shuffle operation during the exchange.

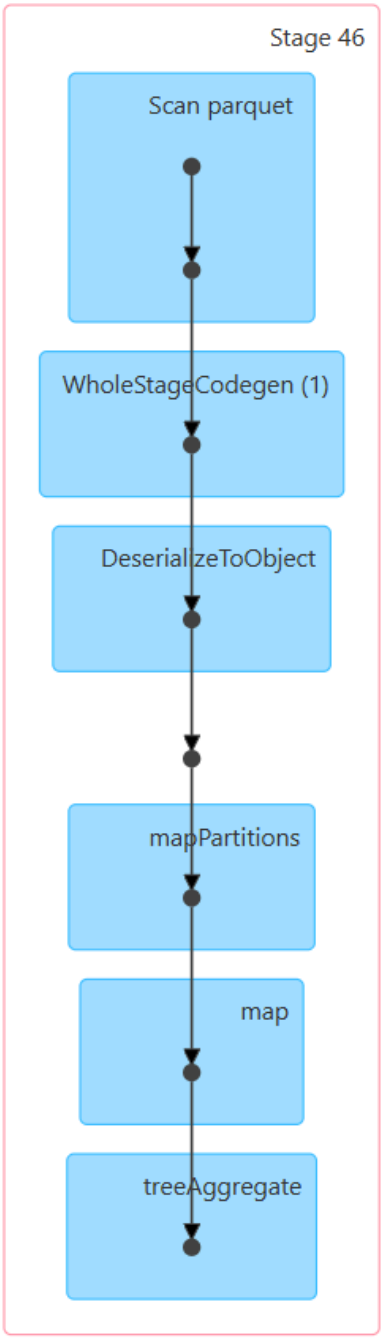


Stage 45: Skipped and Processing

Operations:

1. Exchange: Inputs data from the previous skipped stage.
2. ObjectHashAggregate: Aggregates the data using hashing.
3. WholeStageCodegen (2): Optimizes the stage by combining operations.
4. mapPartitionsInternal: Finalizes computations at the partition level.

Dependencies: The wide dependencies from the Exchange input influence this stage, while later transformations operate independently within partitions.



Stage 46: Complex Transformation with Tree Aggregate

Operations:

1. Scan Parquet: Reads data from Parquet format.
2. WholeStageCodegen (1): Merges multiple transformations into one execution plan.
3. DeserializeToObject: Converts serialized data into objects for manipulation.
4. mapPartitions: Applies transformations at the partition level.
5. map: Further transforms the data.
6. treeAggregate: Performs hierarchical aggregation, reducing computation overhead for large datasets.

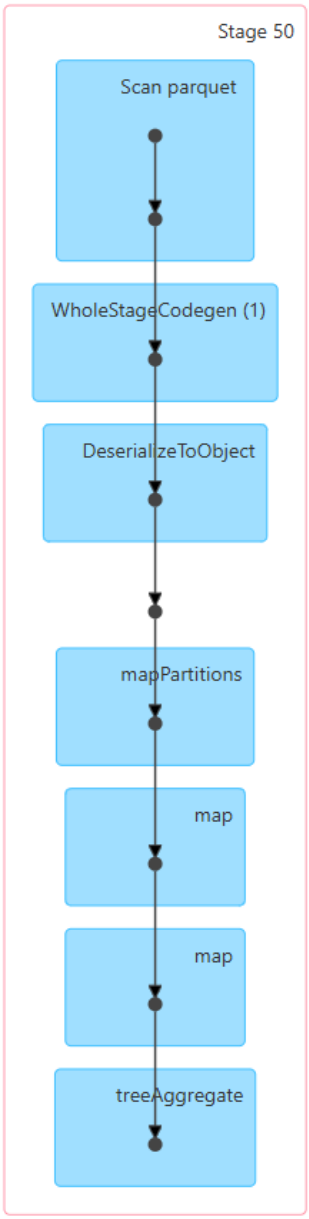
Dependencies: Narrow dependencies exist within map and mapPartitions, while treeAggregate optimizes hierarchical computations.

Stage 50: Extended Transformation with Tree Aggregation

Operations:

1. Scan Parquet: Reads Parquet files for processing.
2. WholeStageCodegen (1): Fuses transformations into one execution plan.
3. DeserializeToObject: Deserializes data for processing.
4. mapPartitions: Applies partition-level transformations.
5. map (2 times): Sequential transformations to refine data.
6. treeAggregate: Performs final hierarchical aggregation.

Dependencies: Narrow dependencies dominate except during scan and hierarchical aggregations.



Stage 51: Scan and Finalize

Operations:

1. Scan Parquet: Reads input data in Parquet format.

2. WholeStageCodegen (1): Optimizes execution by combining operations.
3. mapPartitionsInternal: Final partition-level processing.

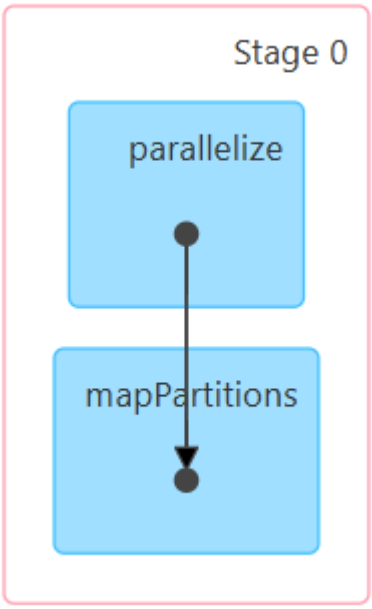
Dependencies: Primarily narrow dependencies, as operations process data within individual partitions

RANDOM FOREST

To proceed with the detailed dependency analysis and regeneration of DAGs for all the uploaded stages, I'll process and outline the steps from the uploaded images, generating descriptions for each and outlining their dependencies.

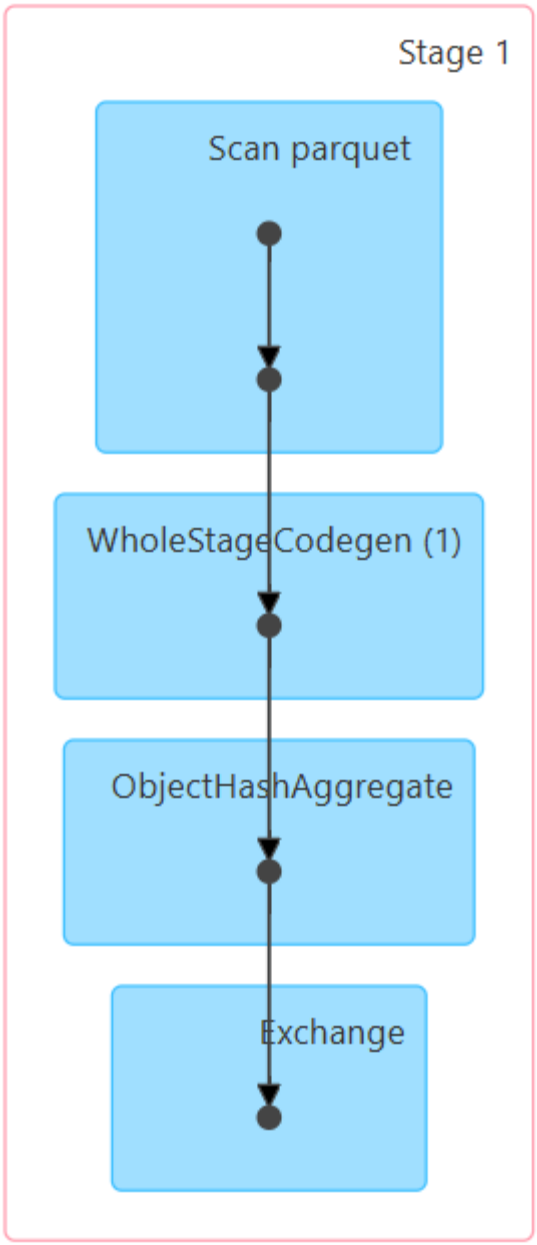
Let me process the latest images to provide the full explanation with added dependencies.

Analysis and Dependencies for Each DAG:



Stage 0

- Components:
  - parallelize: Creates an RDD by distributing a local collection of data across a cluster.
  - mapPartitions: Applies a transformation to each partition of the RDD.
- Dependency:
  - The output of parallelize is consumed by mapPartitions.



Stage 1

Components:

Scan parquet: Reads data from a Parquet file.

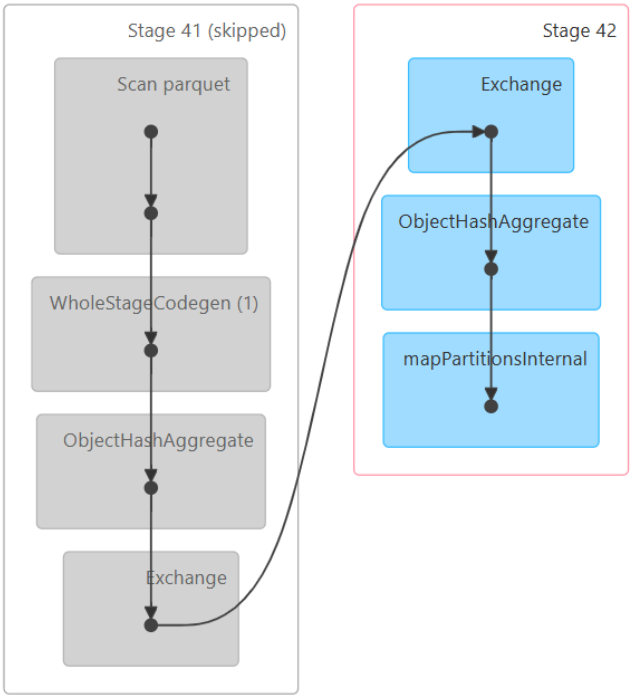
WholeStageCodegen (1): Optimizes the query plan with code generation.

ObjectHashAggregate: Aggregates data using a hash-based mechanism.

Exchange: Transfers data between nodes for distributed processing.

Dependency:

Scan parquet feeds data into WholeStageCodegen (1), which then aggregates it through ObjectHashAggregate before transferring the processed data using Exchange.

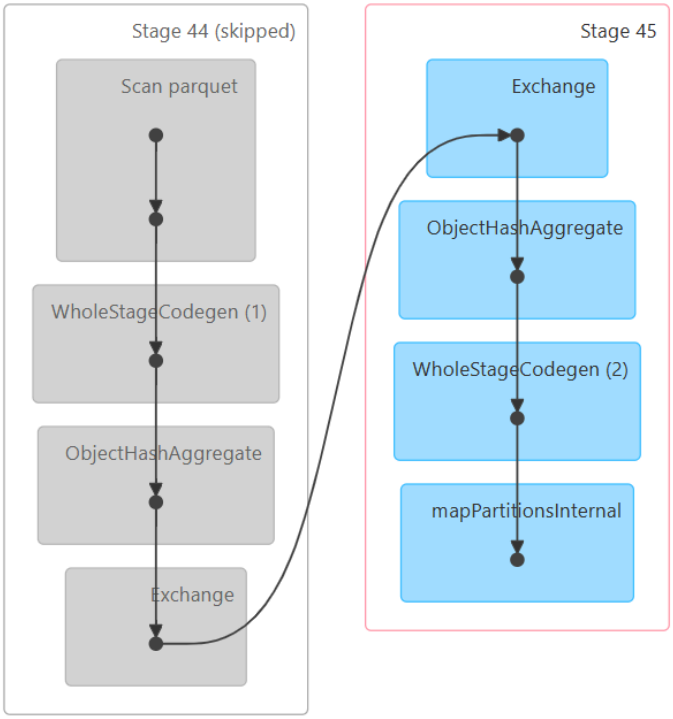


Stage 42
Components:

- Skipped Input Stage: Relies on output from the skipped Stage 41.
- Exchange: Transfers aggregated data.
- ObjectHashAggregate: Continues processing aggregated data.
- mapPartitionsInternal: Applies transformations to partitions within nodes.

Dependency:

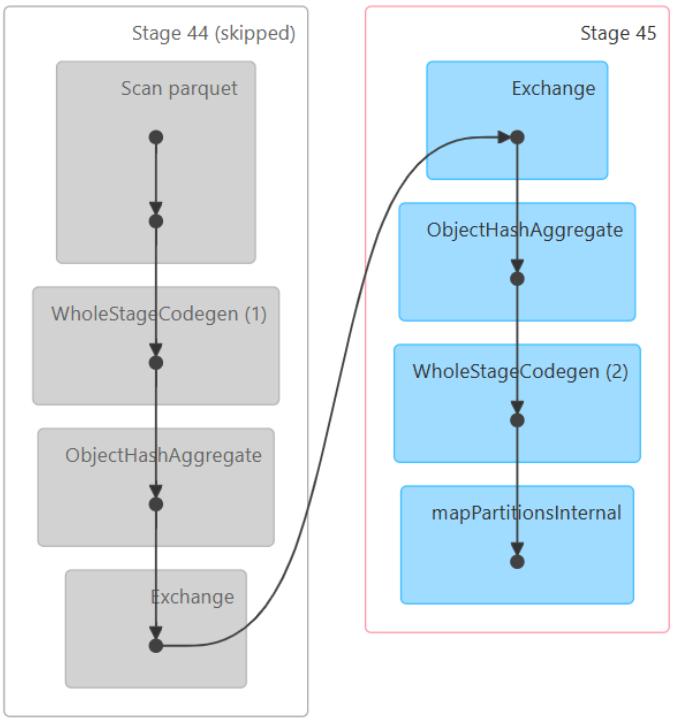
- Exchange relies on aggregated data from Stage 41 and feeds it into ObjectHashAggregate. The result is further transformed by mapPartitionsInternal.



Stage 45
Components:

Skipped Input Stage: Uses output from Stage 44.
Exchange: Transfers aggregated data.
ObjectHashAggregate: Processes data further.
WholeStageCodegen (2): Enhances execution through code generation.
mapPartitionsInternal: Applies transformations within partitions.
Dependency:

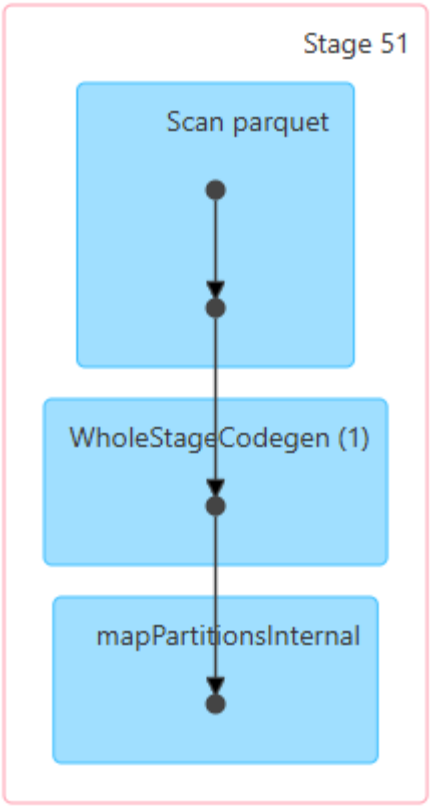
Stage 44 provides input data, processed and transferred through Exchange, aggregated by ObjectHashAggregate, and optimized by WholeStageCodegen (2) before final partition transformations.



Stage 50
Components:

Scan parquet: Reads data from a Parquet file.
WholeStageCodegen (1): Applies query optimizations.
DeserializeToObject: Converts serialized data back to usable objects.
mapPartitions: Applies transformations to partitions.
map (multiple): Applies mapping operations sequentially.
treeAggregate: Combines data using tree aggregation.
Dependency:

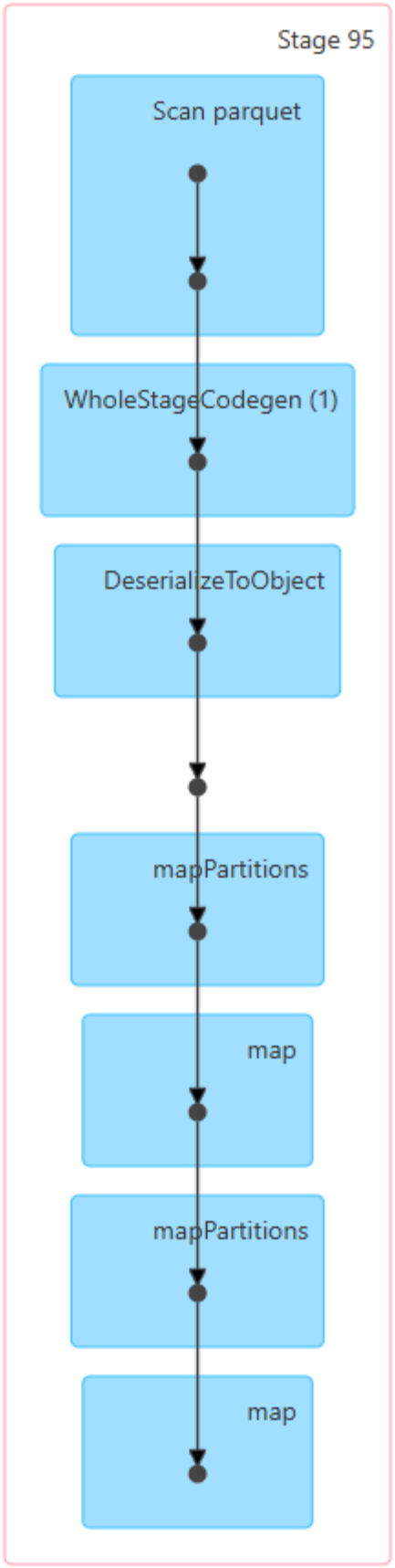
Data flows sequentially from Scan parquet to the final treeAggregate, with transformations and deserializations along the way.



Stage 51
Components:

Scan parquet: Reads data from Parquet.
WholeStageCodegen (1): Optimizes execution.
mapPartitionsInternal: Applies final transformations within partitions.
Dependency:

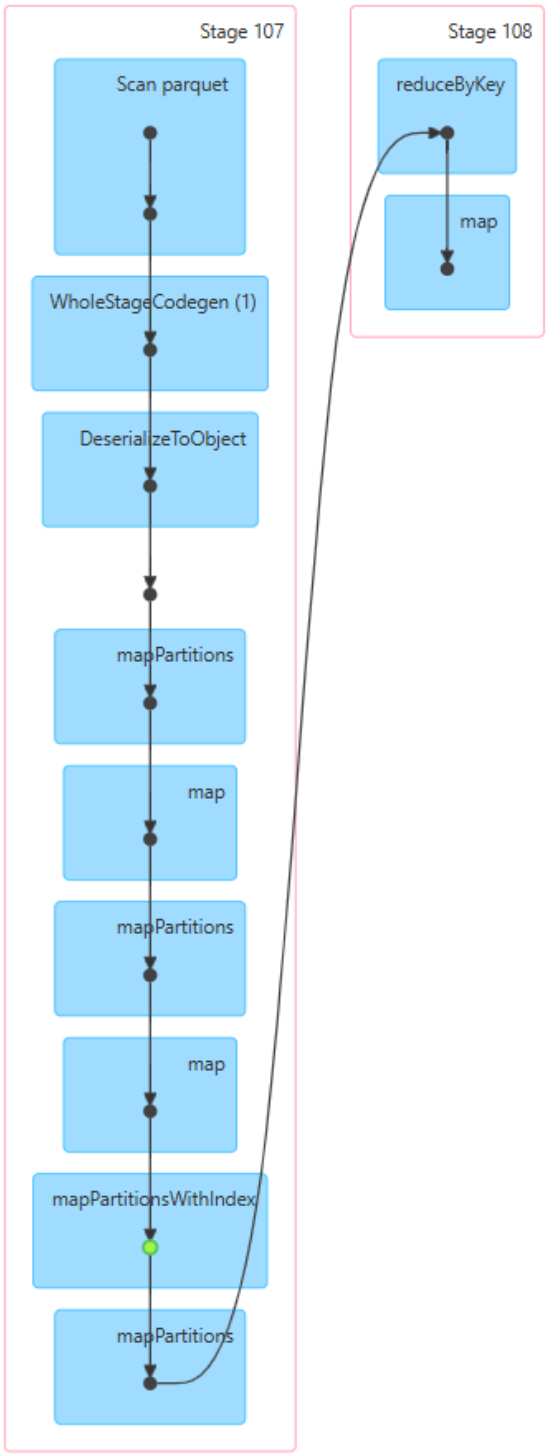
The output from Scan parquet is optimized and transformed by subsequent stages.



Stage 95
Components:

Scan parquet: Reads data.
WholeStageCodegen (1): Optimizes query execution.
DeserializeToObject: Converts serialized data back.
mapPartitions and map: Multiple transformations at different stages.
Dependency:

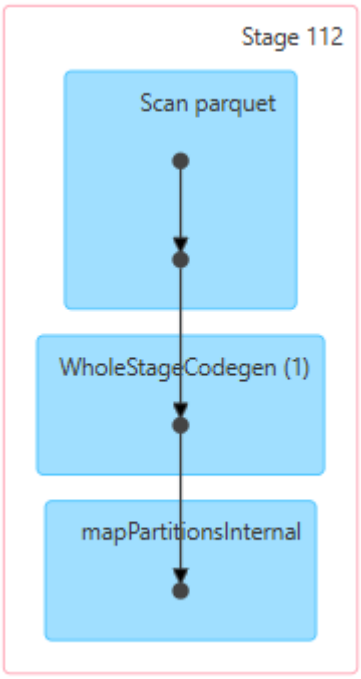
Sequential processing of data from Scan parquet to the final transformation stage.



Stage 108
Components:

Stage 107 input: Provides data to Stage 108.
reduceByKey: Reduces data using key-value pairs.
map: Applies final transformations.
Dependency:

Data from Stage 107 is reduced and mapped for the final output.

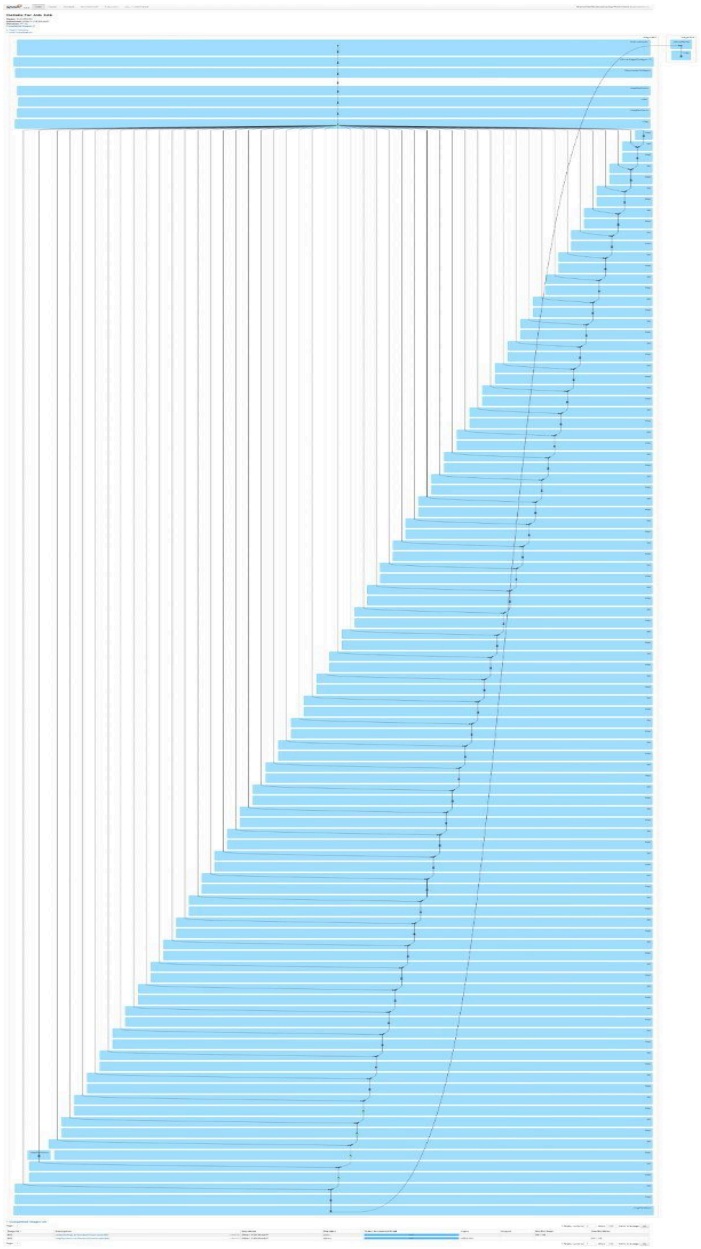


Stage 112
Components:

Scan parquet: Reads data.
WholeStageCodegen (1): Optimizes execution.
mapPartitionsInternal: Applies transformations.
Dependency:

Sequential processing starting with data scanning and ending with transformations.

XGBOOST

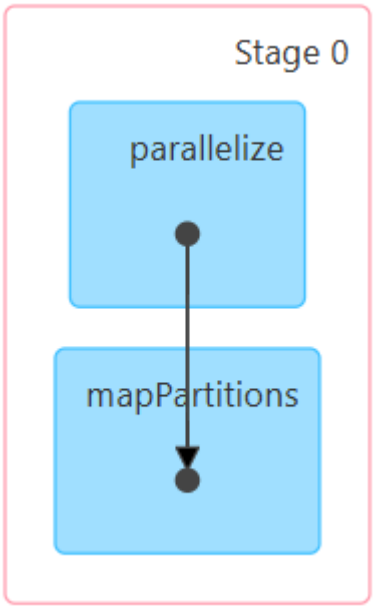


AQEShuffleRead (ShuffledRowRDD):
This stage initiates with an AQE ShuffleRead operation using ShuffledRowRDD, dynamically fetching data shuffled from earlier stages. This balances workload by optimizing partition sizes for efficient computation.
Intermediate Steps (Wide Dependencies):
Wide dependencies are evident, where tasks require data from multiple partitions. These operations involve shuffle steps, necessary for global transformations like joins or aggregations.
Dependency Type:
The dependencies are wide, introducing computational overhead due to shuffle operations and requiring data movement across the cluster.
Optimization Techniques (WholeStageCodegen):
WholeStageCodegen is applied to streamline multiple transformations into a cohesive execution plan. This minimizes intermediate data processing and enhances

performance.

Final Processing (Task Execution):

Tasks are executed efficiently across partitions. The structured distribution ensures even processing, avoiding skew and maximizing resource utilization.



Stage 0

Parallelize Operation:

This stage begins by parallelizing the input data, splitting it into multiple partitions to enable distributed processing.

Intermediate Steps (mapPartitions):

A mapPartitions transformation processes each partition independently. This ensures partition-level computations are optimized for parallel execution.

Dependency Type:

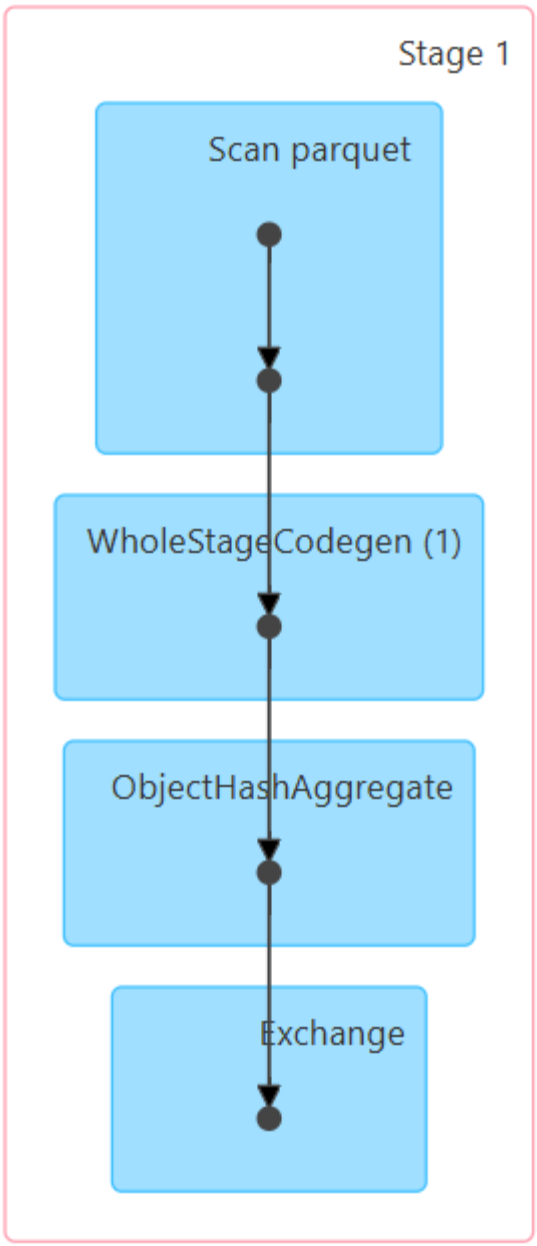
Narrow dependencies dominate, as each task processes data from its respective partition without interdependence.

Optimization Techniques:

The absence of shuffle operations minimizes overhead, while direct partition processing ensures efficient execution.

Final Processing:

Each partition completes its transformation locally, preparing the data for downstream stages.



Stage 1

Scan Parquet:

The stage begins with scanning a Parquet file. This leverages Parquet's columnar storage format, reading only the necessary columns for computation.

Intermediate Steps (WholeStageCodegen & Aggregation):

WholeStageCodegen combines multiple operations (scan, filter, and projection) into a single execution unit.

An ObjectHashAggregate operation follows, performing key-based aggregation using a hash-based grouping mechanism.

Dependency Type:

A wide dependency is introduced during the exchange step, where data is shuffled and repartitioned for the aggregation process.

Optimization Techniques:

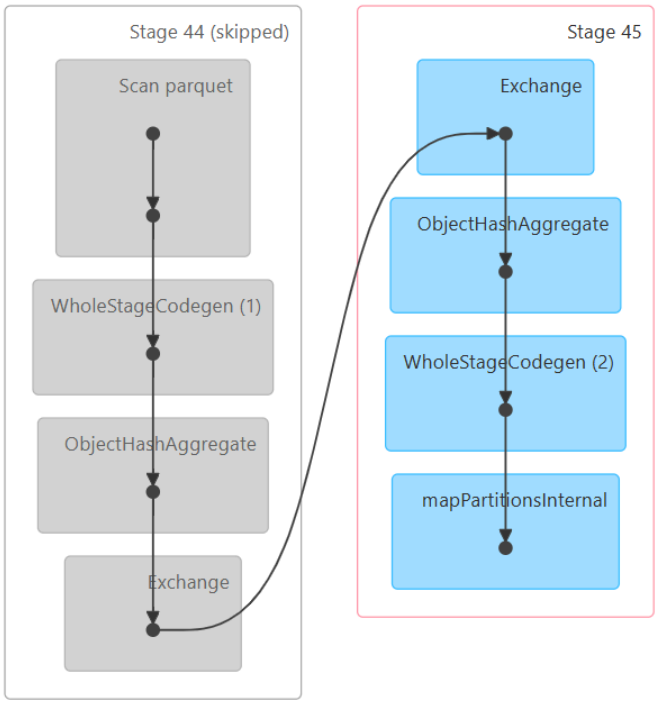
WholeStageCodegen reduces serialization/deserialization costs.

The exchange step optimizes data redistribution for load

balancing.

Final Processing (Exchange):

The stage concludes with an exchange step, redistributing data based on partitioning keys for subsequent computations.



Stages 44 and 45

Exchange (Skipped Stage 44):

Stage 44 is skipped due to caching, avoiding redundant computations. Stage 45 begins with an exchange operation, redistributing cached data across partitions.

Intermediate Steps (WholeStageCodegen & Aggregation):

ObjectHashAggregate computes aggregations efficiently using hash maps.

WholeStageCodegen integrates transformations into a single execution plan, minimizing overhead.

Dependency Type:

A wide dependency arises from the shuffle in the exchange operation, redistributing data for aggregation.

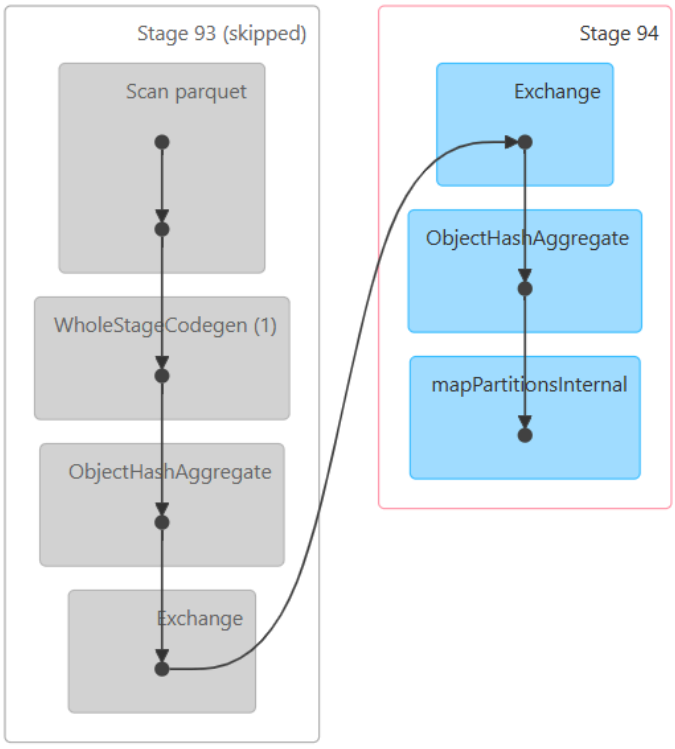
Optimization Techniques:

Caching in Stage 44 reduces I/O overhead.

Combined transformations in Stage 45 enhance processing efficiency.

Final Processing (mapPartitionsInternal):

The stage completes with mapPartitionsInternal, applying computations within each partition to prepare data for downstream tasks.



Stages 93 and 94

Exchange (Skipped Stage 93):

Similar to Stage 44, Stage 93 is skipped, reusing cached data. Stage 94 begins with an exchange operation.

Intermediate Steps (Aggregation & Partition Processing):

ObjectHashAggregate performs hash-based aggregations.

mapPartitionsInternal ensures efficient transformation within partitions.

Dependency Type:

Wide dependencies persist due to shuffle operations, requiring cross-partition data movement.

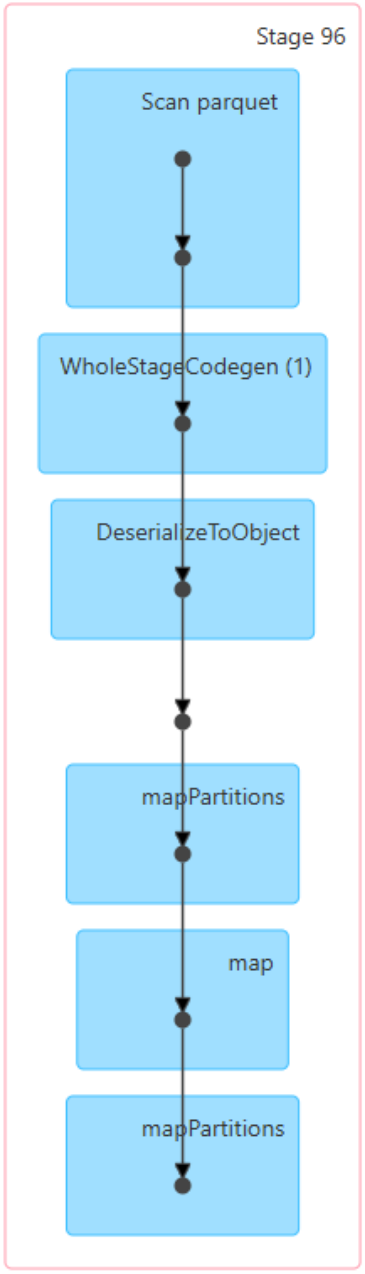
Optimization Techniques:

Reusing cached results avoids recomputation.

Aggregations are optimized with in-memory processing.

Final Processing:

Data is aggregated and redistributed, completing the stage for further processing.



Stage 96

Scan Parquet:

The stage initiates by reading from a Parquet file, utilizing its columnar format for optimized scanning.

Intermediate Steps (Transformation Pipeline):

WholeStageCodegen compiles transformations into a single execution plan.

DeserializeToObject converts serialized data to objects for further transformations.

Dependency Type:

Narrow dependencies dominate, as computations occur within each partition without requiring shuffling.

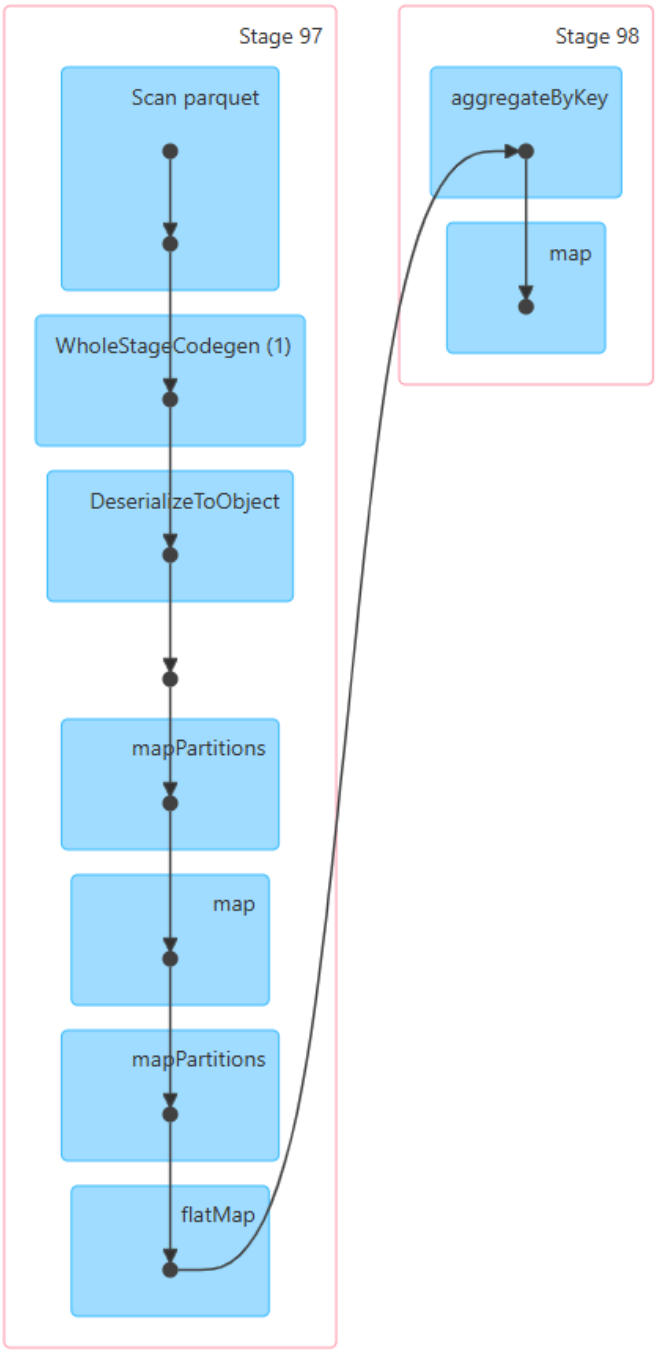
Optimization Techniques:

WholeStageCodegen eliminates redundant operations.

Partition-based computations reduce shuffle overhead.

Final Processing (mapPartitions):

Partition-level transformations complete the stage, producing structured data for downstream stages.



Stages 97 and 98

AggregateByKey (Stage 98):

Stage 97 concludes with a flatMap transformation, preparing data for aggregation in Stage 98. Stage 98 performs an aggregateByKey operation to combine values for each key.

Intermediate Steps (mapPartitions & Aggregation):

mapPartitions transformations process partitioned data.

aggregateByKey aggregates key-based values across partitions.

Dependency Type:

A wide dependency arises during the aggregation, requiring data exchange across partitions.

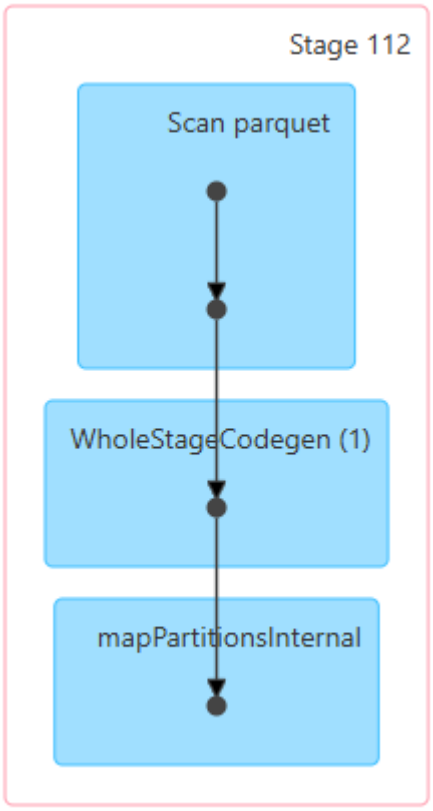
Optimization Techniques:

Efficient key-based aggregation minimizes shuffle overhead.

mapPartitions ensures transformations are performed locally.

Final Processing:

The aggregated data is redistributed, completing the execution plan.



Stage 112

Scan Parquet:

The stage starts by scanning a Parquet file, leveraging its efficient columnar format for data retrieval.

Intermediate Steps (WholeStageCodegen):

WholeStageCodegen optimizes the transformation pipeline by merging operations.

Dependency Type:

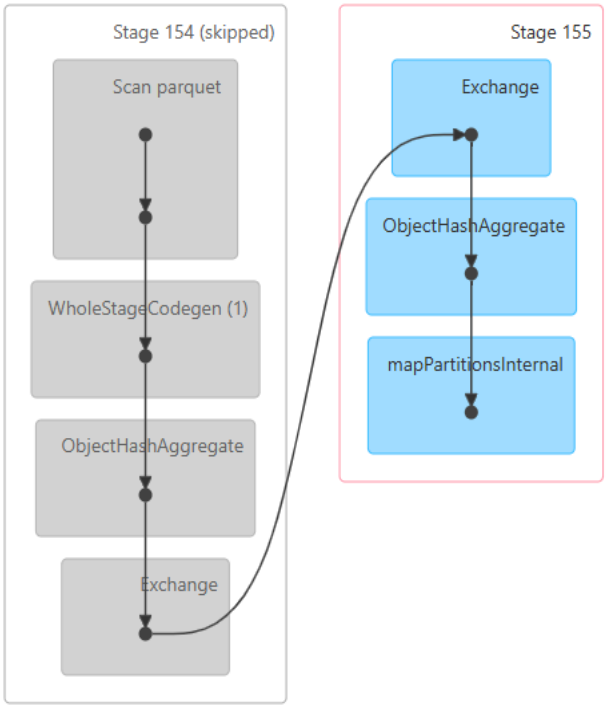
Narrow dependencies allow localized computation within partitions.

Optimization Techniques:

- WholeStageCodegen eliminates intermediate processing overhead.
- Partition-based execution enhances parallelism.

Final Processing (mapPartitionsInternal):

The stage concludes with efficient partition-level transformations, producing structured output for the next stage.



Stages 154 and 155

Exchange (Skipped Stage 154):

Stage 154 is skipped by utilizing cached results. Stage 155 begins with an exchange operation for data redistribution.

Intermediate Steps (Aggregation & Partition Processing):

ObjectHashAggregate performs aggregations.

mapPartitionsInternal applies efficient transformations within partitions.

Dependency Type:

Wide dependencies occur due to data shuffling.

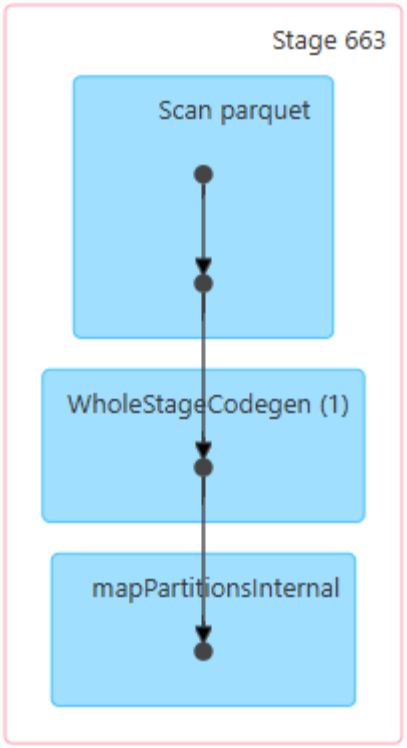
Optimization Techniques:

Caching in Stage 154 minimizes recomputation.

Aggregation logic optimizes memory usage.

Final Processing:

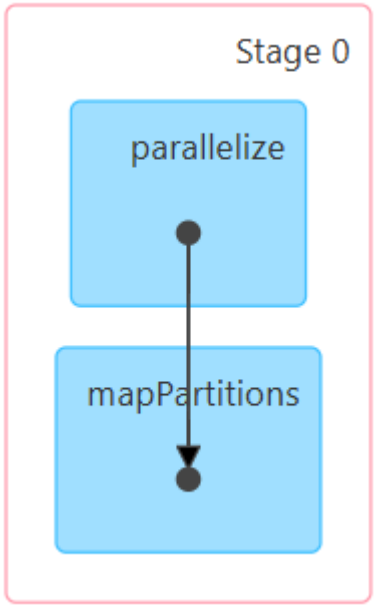
Aggregated and transformed data is redistributed, ready for subsequent stages.



Stage 663

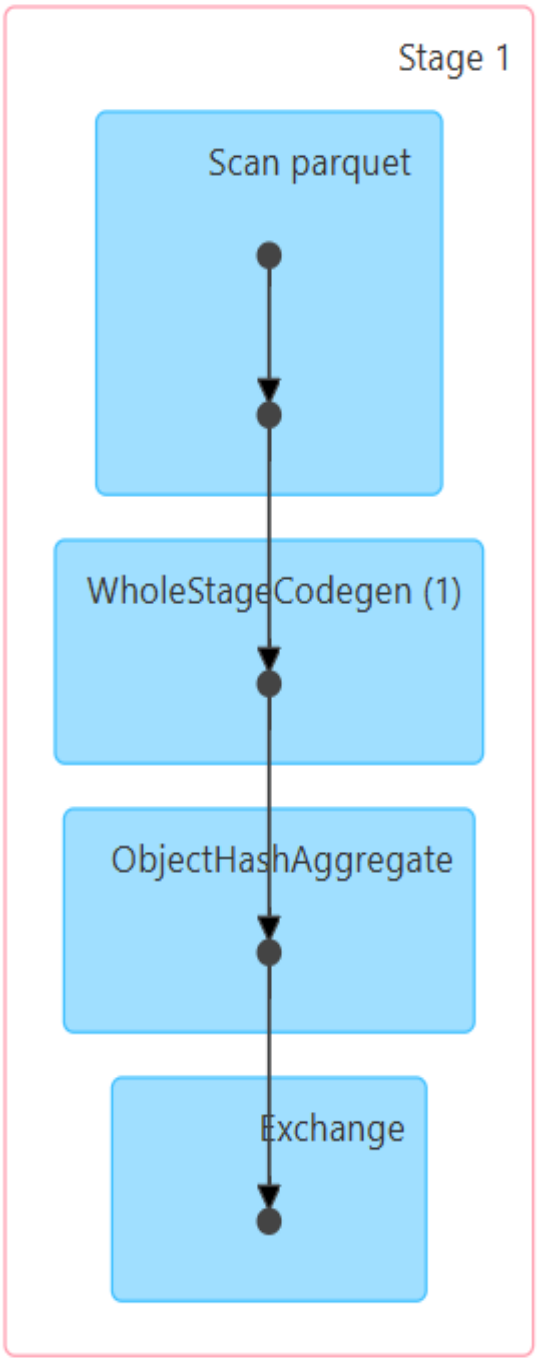
Scan Parquet:
This stage begins by reading data from a Parquet file.
Intermediate Steps (WholeStageCodegen):
WholeStageCodegen optimizes the pipeline by merging transformations into a single execution block.
Dependency Type:
Narrow dependencies dominate, as transformations occur within partitions.
Optimization Techniques:
WholeStageCodegen enhances execution speed by reducing intermediate steps.
Partition-based processing improves parallelism.
Final Processing (mapPartitionsInternal):
The stage completes with efficient partition-level computations, producing data ready for downstream tasks.

Decision tree



Stage 0

Parallelize Operation:
This stage begins by dividing the input data into partitions using the parallelize operation. Each partition represents a slice of the dataset for parallel processing.
Intermediate Steps (mapPartitions):
A mapPartitions transformation processes each partition individually. This step applies specific logic to each partition without affecting others.
Dependency Type:
Narrow Dependency: Each task operates on its respective partition, avoiding inter-partition dependencies.
Optimization Techniques:
Efficient partition-level transformations minimize resource usage.
Avoiding shuffles ensures faster execution.



Stage 1

Scan Parquet:

The execution begins by reading a Parquet file, leveraging its efficient columnar storage for reduced I/O.

Intermediate Steps:

1. WholeStageCodegen (1): Combines multiple logical transformations (e.g., filtering and projections) into a single, optimized execution unit.
2. ObjectHashAggregate: Performs aggregations using a hash-based mechanism, grouping data by keys.
3. Exchange: Redistributes data across partitions, introducing a shuffle.

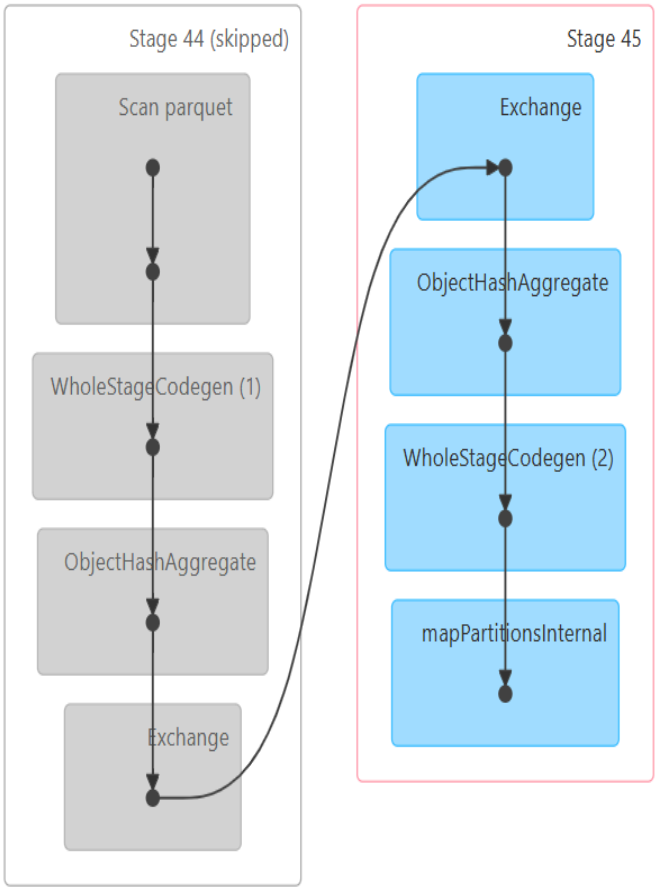
Dependency Type:

Wide Dependency: The shuffle caused by the exchange step requires data transfer across partitions.

Optimization Techniques:

WholeStageCodegen reduces intermediate data materialization.

Efficient aggregation minimizes memory consumption.



Stages 44 and 45

Stage 44 (Skipped):

This stage is skipped due to the presence of cached results from previous computations.

Stage 45:

1. Exchange: Redistributes the cached data across partitions.
2. ObjectHashAggregate: Groups and aggregates the shuffled data.
3. WholeStageCodegen (2): Integrates operations into a single execution pipeline for efficiency.
4. mapPartitionsInternal: Processes each partition independently to finalize transformations.

Dependency Type:

Wide dependencies occur due to the shuffle operation.

Optimization Techniques:

Leveraging cached data reduces redundant I/O operations.

WholeStageCodegen eliminates unnecessary intermediate operations.

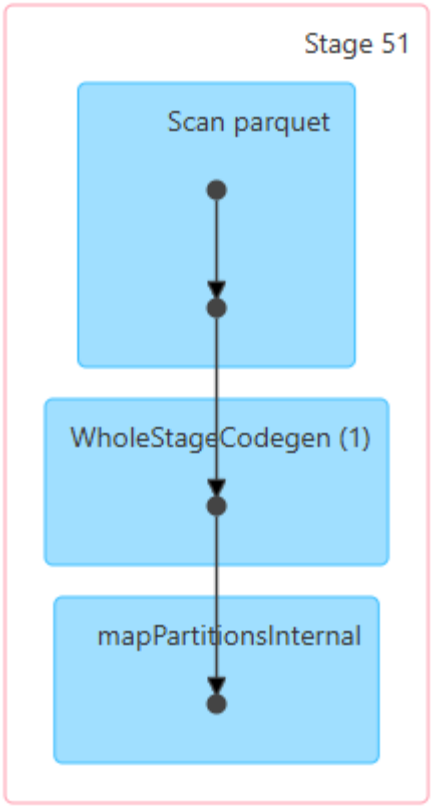
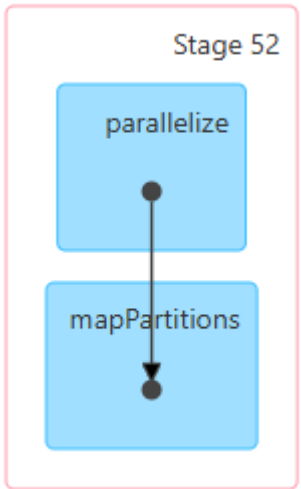


Image 4: Stage 51

Scan Parquet:
Reads data from a Parquet file using Spark's columnar data processing capabilities.
Intermediate Steps:

1. WholeStageCodegen (1): Optimizes the execution of transformations by merging them into a single code block.
2. mapPartitionsInternal: Applies transformation logic within partitions.

Dependency Type:
Narrow dependencies ensure localized processing without shuffles.
Optimization Techniques:
Combining operations reduces overhead.
Partition-local computations improve performance.



Stage 52

Parallelize Operation:
Divides input data into partitions for parallel processing.
Intermediate Steps (mapPartitions):
Each partition undergoes independent transformation logic using mapPartitions.
Dependency Type:
Narrow dependencies ensure that tasks only process their respective partitions.
Optimization Techniques:
Minimal overhead due to the absence of shuffles.
Partition-level processing optimizes resource usage.

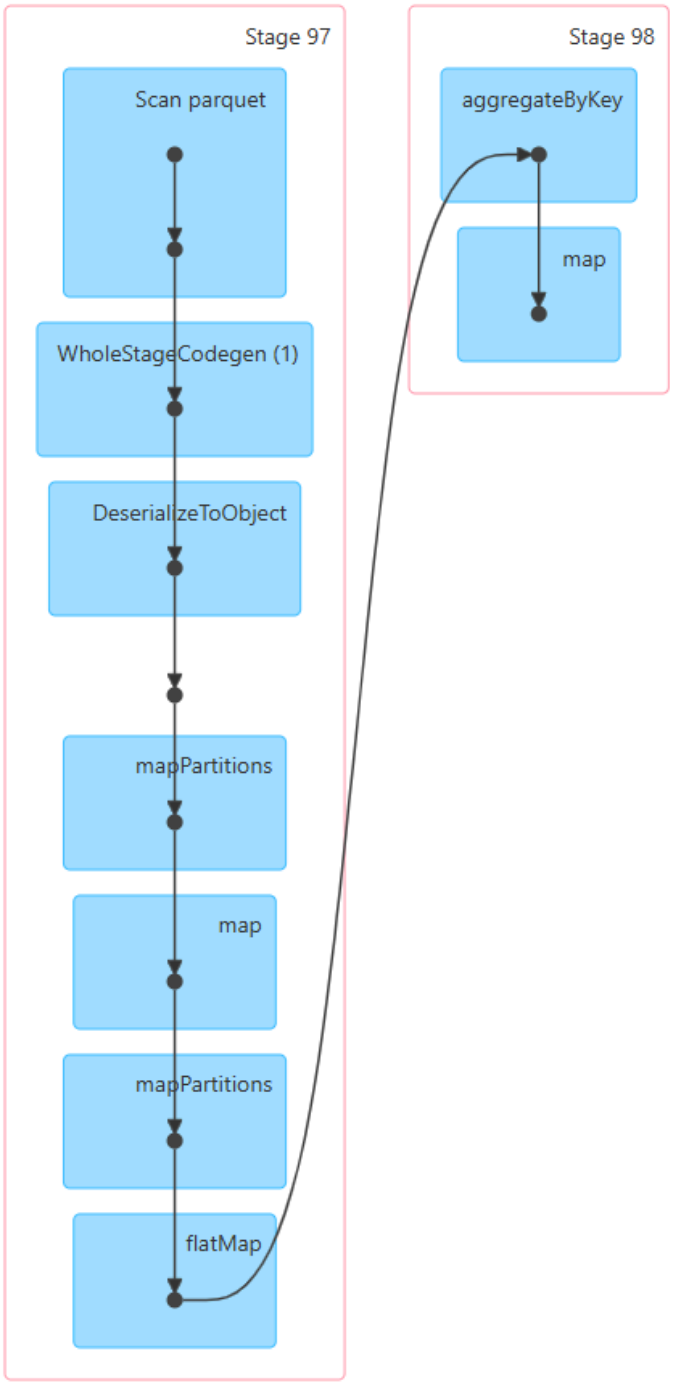


Image 6: Stages 97 and 98

Stage 97:

1. Scan Parquet: Reads data using Spark's optimized Parquet scan.
2. WholeStageCodegen (1): Combines

transformations into a single execution plan.

3. DeserializeToObject: Converts serialized data to Spark objects.
4. mapPartitions: Applies transformations partition by partition.
5. flatMap: Flattens the output to prepare for key aggregation.

Stage 98:

1. aggregateByKey: Aggregates values for the same key across partitions.
2. map: Applies final transformation logic.

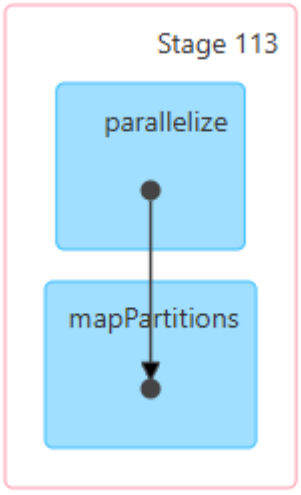
Dependency Type:

Wide dependencies in aggregateByKey due to shuffle operations.

Optimization Techniques:

WholeStageCodegen minimizes intermediate materialization.

Efficient key aggregation reduces memory overhead.



Stage 113

Parallelize Operation:

Creates parallel partitions from the input dataset.

Intermediate Steps (mapPartitions):

Applies transformation logic on each partition independently.

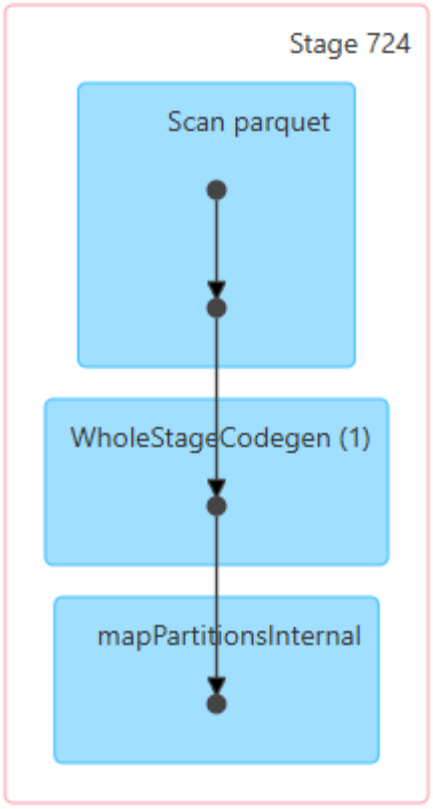
Dependency Type:

Narrow dependencies ensure efficient localized processing.

Optimization Techniques:

Avoiding inter-partition communication reduces execution time.

Efficient resource allocation during transformations.



Stage 724

Scan Parquet:

Reads data using an optimized scan of a Parquet file.

Intermediate Steps:

1. WholeStageCodegen (1): Merges logical operations for faster execution.
2. mapPartitionsInternal: Finalizes transformation within partitions.

Dependency Type:

Narrow dependencies ensure efficient, localized processing.

Optimization Techniques:

WholeStageCodegen reduces processing latency.

Partition-local computations enhance parallelism.

Comprehensive Execution Plan

Detailed Observations:

The execution plan displays a mix of narrow and wide dependencies.

Shuffle Operations: Introduced at stages with wide dependencies, redistributing data for tasks like joins or aggregations.

Task Distribution: Tasks are evenly spread across the cluster, ensuring balanced load.

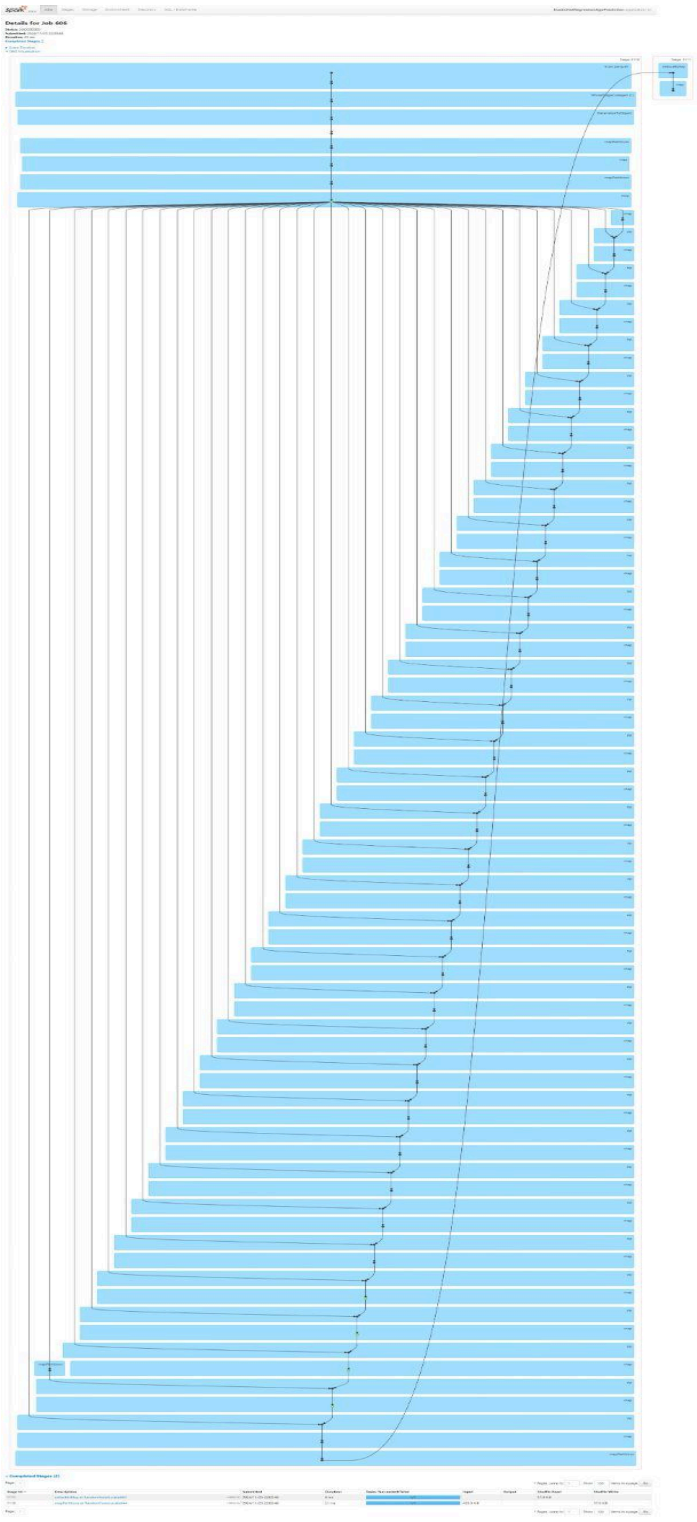
Optimization Techniques:

Caching reduces redundant computations.

WholeStageCodegen minimizes execution overhead for transformation pipelines.

Balanced partitioning prevents skew, ensuring efficient execution across the cluster.

Final Observations: The overall job is optimized for distributed execution, leveraging Spark's capabilities to process large datasets with minimal overhead.



AQEShuffleRead (ShuffledRowRDD):

The execution plan begins with an Adaptive Query Execution (AQE) ShuffleRead operation utilizing ShuffledRowRDD. This step dynamically retrieves data shuffled from previous stages, ensuring partition sizes are optimized to balance the workload across the cluster for efficient processing.

Intermediate Steps (MapPartitionsRDD):

The shuffled data is processed by a MapPartitionsRDD at the partition level. This step structures and organizes the data for efficient execution in subsequent computations. Partition-level processing ensures that only relevant data is processed, reducing unnecessary overhead.

Dependency Type:

This stage showcases wide dependencies, where partitions rely on data from multiple other partitions. Such dependencies are introduced during operations like groupByKey, join, or aggregateByKey. These are critical for global transformations but add computational and I/O overhead due to shuffle operations.

Optimization Techniques (WholeStageCodegen):

Spark employs WholeStageCodegen to fuse multiple transformations into a single, optimized execution pipeline. By doing so, it reduces intermediate operations, minimizing serialization and deserialization costs while improving overall performance. This approach ensures that CPU cycles are spent on meaningful computations rather than overhead tasks.

Final Processing (mapPartitionsInternal):

The stage concludes with the mapPartitionsInternal operation. This step executes efficient computations within each partition, completing all necessary transformations before transitioning to the next stage. The focus on partition-local processing ensures that results are prepared with minimal data movement across the cluster.

Summary: This stage effectively utilizes AQE, optimized partitioning, and WholeStageCodegen to streamline execution. While wide dependencies introduce computational overhead, the use of partition-local transformations (via mapPartitionsInternal) ensures efficient and balanced workload distribution, preparing the data for subsequent processing.

VII. Conclusion

In conclusion, the transition from traditional machine learning models in Phase 2 to distributed processing using PySpark in Phase 3 highlighted a trade-off between error metrics and model performance. While models in Phase 3 exhibited increased Mean Absolute Error (MAE) and Mean Squared Error (MSE), the R^2 values showed consistent improvement, indicating that distributed processing enhanced the models' ability to explain variance in the data. This was particularly evident in complex models like Random Forest and Gradient Boosting, where the scalability and computational resources of PySpark led to better explanatory power. Overall, while error rates increased, the distributed approach provided significant advantages in handling large datasets and improving model performance, especially for complex models where accuracy is critical.

References or data source

[1] M abdullah. (2024). Human Age Prediction Synthetic Dataset [Data set]. Kaggle. https://doi.org/10.34740/KAGGLE/DSV/931458

[2] Montgomery, D. C., Peck, E. A., & Vining, G. G. (2012). Introduction to Linear Regression Analysis. Wiley. doi:10.1002/9781118490587

[3] Smola, A. J., & Schölkopf, B. (2004). "A tutorial on support vector regression." Statistics and Computing, 14(3), 199-222. doi:10.1023/B.0000035301.49549.88

[4] Cover, T. M., & Hart, P. E. (1967). "Nearest Neighbor Pattern Classification." IEEE Transactions on Information Theory, 13(1), 21-27. doi:10.1109/TIT.1967.1053964

[5] Breiman, L. (2001). "Random Forests." Machine Learning, 45(1), 5-32. doi:10.1023/A:1010933404324

[6] Chen, T., & Guestrin, C. (2016). "XGBoost: A Scalable Tree Boosting System." In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (pp. 785-794). doi:10.1145/2939672.2939785

[7] Haykin, S. (1998). Neural Networks: A Comprehensive Foundation. Prentice Hall. ISBN: 978-0-02-052420-2

[8] Breiman, L., Friedman, J. H., Olshen, R. A., & Stone, C. J. (1986). Classification and Regression Trees. Wadsworth and Brooks/Cole. ISBN: 978-0-534-19638-8

[9] Friedman, J. H. (2001). "Greedy Function Approximation: A Gradient Boosting Machine." Annals of Statistics, 29(5), 1189-1232. doi:10.1214/aos/1013203451

[10] C. O'Neill and R. Schutt. Doing Data Science., O'Reilly. 2013

[11] Kevin Murphy,"Machine Learning: A Probabilistic Perspective", MIT Press, 2012.

Peer Evaluation Form for Final Group Work

CSE 487/587B

- Please write the names of your group members.

Group member 1 : Prathamesh Ravindra Varhadpande

Group member 2 : Nikhil Gokhale

Group member 3 : Nachiket Dabhade

- Rate each groupmate on a scale of 5 on the following points, with 5 being HIGHEST and 1 being LOWEST.

Evaluation Criteria	Group member 1	Group member 2	Group member 3
How effectively did your group mate work with you?	5	5	5
Contribution in writing the report	5	5	5
Demonstrates a cooperative and supportive attitude.	5	5	5
Contributes significantly to the success of the project .	5	5	5
TOTAL	20	20	20

Also please state the overall contribution of your teammate in percentage below, with total of all the three members accounting for 100% (33.33+33.33+33.33 ~ 100%) :

Group member 1 : 33.33

Group member 2 : 33.33

Group member 3 : 33.33