LOW LEVEL DOCUMENTATION -1

PROJECT-1

**Creation of S3 bucket:**

To go on with the project First we have created a S3 Bucket with the name groupno6 then we have created the folders like:

* Data/
* Logs/
* Results/
* Temp\_dir/

And the **subfolders** like:

* Data/cricket\_databases/
* Data/cricket\_databases/cricket\_csv
* Data/cricket\_databases/cricket\_parquet
* Data/cricket\_databases/cricket\_csv/data=20230608/
* Data/cricket\_databases/cricket\_csv/data=20230608/IPL\_Ball\_by\_Ball\_2008\_2022.csv
* Data/cricket\_databases/cricket\_csv/data=20230608/IPL\_Matches\_2008\_2022.csv

**Creation of IAM Role:**

Create an Admin full access IAM role “Glue\_AdminFullAccessRole”

* 1. Go to IAM à Roles à Create role
  2. Select Type of trusted entity: AWS Service àService=Glue à Next
  3. Search policy à AdministratorAccess à Next:Tags à Next:Review
  4. Role Name: Glue\_AdminFullAccessRole à Create Role

**Creation of Database:**

We have created a database named **groupno6** and added tables using crawler / we can directly add the tables from S3 location with the names

* ipl\_ball\_by\_ball\_2008\_2022\_csv
* ipl\_matches\_2008\_2022\_csv

Create glue job with the script:

**EXPLANATION:**

* **SparkSession.builder.appName("AWS Project 1"):** This creates a builder object for creating a SparkSession, which is the entry point for interacting with Spark. The appName function sets the name of the Spark application to "AWS Project 1".
* **.getOrCreate():** This method retrieves an existing SparkSession or creates a new one if it doesn't exist. It ensures that only one SparkSession is created per JVM (Java Virtual Machine).
* **args = getResolvedOptions(sys.argv, ["JOB\_NAME"]):** This line uses the getResolvedOptions function from AWS Glue to retrieve the resolved command-line arguments for the Glue job. It takes sys.argv (the list of command-line arguments) and a list of argument names to retrieve as input and returns a dictionary of the resolved options.
* **glueContext = GlueContext(spark.sparkContext**): This creates a GlueContext object, which is the entry point for interacting with AWS Glue. It takes the SparkContext from the SparkSession (spark.sparkContext) as input.
* **job = Job(glueContext):** This creates a Job object from the GlueContext. The Job object represents the AWS Glue job that will be executed.
* **job.init(args["JOB\_NAME"], args):** This initializes the Glue job by calling the init method on the Job object. It takes the value of the "JOB\_NAME" argument from the args dictionary as the job name and the args dictionary itself as the resolved options for the job.
* The **‘match file’** will contain the path of the file where the IPL\_Matches\_2008\_2022.csv is located
* The **‘ball file’** will contain the path of the file where the IPL\_Ball\_by\_Ball\_2008\_2022.csv is located
* The output path will contain the path where the result should be stored
* **spark.read.format("csv").option("header", "true").load(match\_file):** This reads the CSV file located at match\_file using the SparkSession's read method. The format("csv") specifies that the data is in CSV format. The option("header", "true") indicates that the first row of the CSV file contains the column headers. The load(match\_file) loads the CSV file into a DataFrame.
* **.withColumnRenamed("ID", "MATCH\_ID"):** This renames the column named "ID" in the DataFrame to "MATCH\_ID" using the withColumnRenamed method. The resulting DataFrame is assigned to the variable match.
* **spark.read.format("csv").option("header", "true").load(ball\_file):** This reads the CSV file located at ball\_file using the same approach as described above. The resulting DataFrame is assigned to the variable ball.
* **match\_ball = ball.join(f.broadcast(match), ball.ID == match.MATCH\_ID, "inner"):** This performs an inner join operation on the ball DataFrame and the match DataFrame using the "ID" column from ball and the "MATCH\_ID" column from match. The resulting DataFrame is assigned to the variable match\_ball.
* **match\_ball.createOrReplaceTempView("ALL"):** This creates a temporary view named "ALL" for the match\_ball DataFrame. A temporary view allows you to query the DataFrame using SQL-like syntax.

**REQUIREMENT 1:**

* The Requirement 1 is to calculate the **“Fastest Centuries”**
* We are using SQL Query

**ETL STEPS:**

* ROW\_NUMBER() OVER(ORDER BY ROUND(100\*(SUM(BATSMAN\_RUN)/COUNT(BATSMAN\_RUN)), 2) ASC) AS POS: It calculates the row number for each player based on the batting average (runs per innings) and orders the rows in ascending order.
* SUM(CASE WHEN EXTRA\_TYPE IN ("wides", "byes", "legbyes") THEN 0 ELSE BATSMAN\_RUN END) AS Runs: Calculates the total runs for each player, excluding runs from wides, byes, and leg byes.
* SUM(CASE WHEN EXTRA\_TYPE IN ("NA") THEN 1 ELSE 0 END) AS BF: Calculates the number of balls faced by each player, excluding NAs.
* SUM(CASE WHEN BATSMAN\_RUN = 4 AND NON\_BOUNDARY = 0 THEN 1 ELSE 0 END) AS 4s``: Calculates the number of 4s (fours) hit by each player, excluding non-boundaries.
* SUM(CASE WHEN BATSMAN\_RUN = 6 AND NON\_BOUNDARY = 0 THEN 1 ELSE 0 END) AS 6s``: Calculates the number of 6s (sixes) hit by each player, excluding non-boundaries.
* CASE WHEN BATTINGTEAM != TEAM1 THEN TEAM1 WHEN BATTINGTEAM != TEAM2 THEN TEAM2 END AS Against: Determines the opposing team for each player based on the batting team.
* ans = spark.sql(query).coalesce(1).dropna(): Executes the SQL query using Spark SQL and retrieves the result. The coalesce(1) ensures that the result is written into a single partition, and dropna() removes any rows with null values.
* window\_spec = Window.partitionBy("SEASON").orderBy(f.asc("Runs")): Defines a window specification partitioned by the "SEASON" column and ordered by ascending "Runs".
* ans = ans.withColumn("POS", f.row\_number().over(window\_spec)): Adds a new column "POS" to the DataFrame, which represents the row number within each season based on the "Runs" column ordering.

**REQUIREMENT 2:**

* We Are Using DATAFRAMES in this requirement
* The s3bucket\_nodel will contain the address of ipl\_ball\_by\_ball\_2008\_2022\_csv
* The Amazons3\_nodel will contain the address of ipl\_matches\_2008\_2022\_csv
* The ApplyMapping\_node2 will contain all the columns present in ‘ipl\_ball\_by\_ball\_2008\_2022\_csv’

**ETL STEPS:**

* The ChangeSchema\_node1 will contain all the columns present in ‘ipl\_matches\_2008\_2022\_csv’
* It reads the data from the two datasets (S3bucket\_node1 and AmazonS3\_node1686216371280) using the glueContext.create\_dynamic\_frame.from\_catalog function.
* The script applies mapping to the data using the ApplyMapping.apply function for both datasets, specifying the source and target columns.
* The script joins the two datasets based on the "id" column using the join method.
* It filters out rows where the "extra\_type" column is not equal to 'wides'.
* The script selects specific columns from the joined and filtered dataset to create a new DataFrame.
* It performs grouping and aggregation on the selected DataFrame to calculate various statistics such as total runs, boundaries (4s and 6s), and strike rate.
* The script renames and transforms certain columns in the DataFrame to make them more readable and meaningful.
* The DataFrame is ordered in descending order based on the number of boundaries (4s).
* It adds a new column for the position (POS) within each season using the row\_number function and windowing.
* The final DataFrame is converted back to a DynamicFrame using DynamicFrame.fromDF.
* Finally, the result is written to an S3 bucket in CSV format using the glueContext.write\_dynamic\_frame.from\_options function.

**REQUIREMENT 3:**

* The Requirement 1 is to calculate the **“Most Runs”**
* The **‘match file’** will contain the path of the file where the IPL\_Matches\_2008\_2022.csv is located
* The **‘ball file’** will contain the path of the file where the IPL\_Ball\_by\_Ball\_2008\_2022.csv is located

**ETL STEPS:**

* match=spark.read.format("csv").option("header", "true").load(match\_file): This line reads the match data CSV file using the CSV format and sets the "header" option to true, indicating that the first row contains column names. The load(match\_file) part specifies the file path. The resulting DataFrame is assigned to the variable match.
* .withColumnRenamed("ID", "MATCH\_ID"): This part of the code renames the column "ID" in the match DataFrame to "MATCH\_ID" using the withColumnRenamed() function. The updated DataFrame is still assigned to the variable match.
* ball=spark.read.format("csv").option("header", "true").load(ball\_file): Similar to the previous line, this code reads the ball data CSV file using the CSV format and sets the "header" option to true. The resulting DataFrame is assigned to the variable ball.
* match\_ball=ball.join(f.broadcast(match), ball.ID==match.MATCH\_ID, "inner"): This line performs an inner join operation between the ball and match DataFrames. The join condition is specified as ball.ID==match.MATCH\_ID, which means the join is performed when the "ID" column in the ball DataFrame matches the "MATCH\_ID" column in the match DataFrame. The "inner" argument indicates that only matching records should be included in the result.
* The f.broadcast() function is used to optimize the join operation by broadcasting the match DataFrame to all the worker nodes.
* match\_ball.createOrReplaceTempView("ALL"): After performing the join, the resulting DataFrame (match\_ball) is registered as a temporary view named "ALL". This allows you to use SQL queries to manipulate and analyze the combined data.
* In a query we use CTE’s because separating calculation into a CTE, the main query can then aggregate the results from the "DATA" CTE to obtain the final desired output. The CTE starts with a SELECT statement that retrieves specific columns from the "ALL" table, including BATTER, ID, and SEASON. It also includes additional calculations and aggregations.
* The query begins with the SELECT statement, which specifies the columns to be included in the result set.
* **ROW\_NUMBER() OVER(ORDER BY SUM(RUNS) DESC) AS P** assigns a sequential number (P) to each row in the result set based on the descending order of the total runs scored (SUM(RUNS)). This provides a ranking of the players based on their total runs.
* **BATTER as Player** renames the BATTER column as "Player" in the output.
* **SUM(MATCHES) AS Mat calculates** the total number of matches (Mat) played by each player by summing up the MATCHES column from the "DATA" CTE.
* **SUM(INNINGS) AS Inns** calculates the total number of innings (Inns) played by each player by summing up the INNINGS column from the "DATA" CTE.
* **SUM(NO) AS NO** calculates the total number of times (NO) a player was not out by summing up the NO column from the "DATA" CTE.
* **SUM(RUNS) AS Runs** calculates the total runs scored (Runs) by summing up the RUNS column from the "DATA" CTE.
* **MAX(RUNS) AS HS** calculates the highest individual score (HS) by taking the maximum value from the RUNS column.
* **ROUND((SUM(RUNS)/(SUM(INNINGS)-SUM(NO))), 2) AS AVG** calculates the batting average (AVG) by dividing the total runs by the total number of innings minus the number of times the player was not out. The result is rounded to two decimal places.
* **SUM(BF)** BF calculates the total number of balls faced (BF) by summing up the BF column from the "DATA" CTE.
* **CAST((SUM(RUNS)/SUM(BF))\*100 AS DECIMAL(10, 2)) AS SR** calculates the batting strike rate (SR) by dividing the total runs by the total balls faced, multiplying by 100, and converting the result to a decimal with two decimal places.
* **SUM(CS) AS CS** calculates the total number of centuries (CS) by summing up the CS column from the "DATA" CTE.
* **SUM(FS) AS FS** calculates the total number of half-centuries (FS) by summing up the FS column from the "DATA" CTE.
* **SUM(FOUR) AS FOUR** calculates the total number of fours (FOUR) hit by summing up the FOUR column from the "DATA" CTE.
* **SUM(SIX) AS SIX** calculates the total number of sixes (SIX) hit by summing up the SIX column from the "DATA" CTE.
* **SEASON** specifies that the grouping should be done based on the SEASON column.
* **GROUP BY BATTER, SEASON** groups the data based on the unique combinations of "BATTER" and "SEASON", ensuring that the aggregation functions are applied to the appropriate groups.
* Overall, the main query aggregates the data from the "DATA" CTE and calculates various statistics and metrics for each player, such as the total runs, batting average, strike rate.

**REQUIREMENT 4:**

* The Requirement 1 is to calculate the “Most Wickets”
* The ‘match file’ will contain the path of the file where the IPL\_Matches\_2008\_2022.csv is located
* The ‘ball file’ will contain the path of the file where the IPL\_Ball\_by\_Ball\_2008\_2022.csv is located
* **ETL STEPS:**
* match=spark.read.format("csv").option("header", "true").load(match\_file): This line reads the match data CSV file using the CSV format and sets the "header" option to true, indicating that the first row contains column names. The load(match\_file) part specifies the file path. The resulting DataFrame is assigned to the variable match.
* .withColumnRenamed("ID", "MATCH\_ID"): This part of the code renames the column "ID" in the match DataFrame to "MATCH\_ID" using the withColumnRenamed() function. The updated DataFrame is still assigned to the variable match.
* ball=spark.read.format("csv").option("header", "true").load(ball\_file): Similar to the previous line, this code reads the ball data CSV file using the CSV format and sets the "header" option to true. The resulting DataFrame is assigned to the variable ball.
* The ball DataFrame is joined with the match DataFrame using a column comparison between ball.ID and match.MATCH\_ID. The join type is specified as "inner", which means only the matching rows from both DataFrames will be included in the result. The f.broadcast() function is used to optimize the join operation by broadcasting the match DataFrame to all the worker nodes.
* The resulting DataFrame from the join operation is assigned to the variable match\_ball.
* Three consecutive filter operations are applied to the match\_ball DataFrame using the filter() function. Each filter excludes specific values from the extra\_type column. Rows where extra\_type is equal to 'byes', 'legbyes', or 'penality' are filtered out from the DataFrame.
* After the filtering operations, the match\_ball DataFrame is registered as a temporary view named "ALL" using the createOrReplaceTempView() function. This allows the DataFrame to be referenced by its view name in subsequent SQL queries.
* In a query we use CTE’s because separating calculation into a CTE, the main query can then aggregate the results from the "DATA" CTE to obtain the final desired output. The CTE starts with a SELECT statement that retrieves specific columns from the "ALL" table, including BATTER, ID, and SEASON. It also includes additional calculations and aggregations.
* After the CTE, a SELECT statement is used to retrieve the final result. It selects the following columns:
* **BOWLER:** The name of the bowler.
* **SUM(MATCHES) as MATCHES:** Sums the MATCHES column from the DATA CTE, representing the total number of matches played by the bowler.
* **SUM(INNINGS) as INNINGS:** Sums the INNINGS column from the DATA CTE, representing the total number of innings played by the bowler.
* **SUM(WICKETS) as WICKETS:** Sums the WICKETS column from the DATA CTE, representing the total number of wickets taken by the bowler.
* **SUM(RUNS) as RUNS:** Sums the RUNS column from the DATA CTE, representing the total runs conceded by the bowler.
* **SUM(OVERS) as OVERS:** Sums the OVERS column from the DATA CTE, representing the total number of overs bowled by the bowler.
* **ROUND((SUM(RUNS)/SUM(OVERS)), 2) as ECON:** Calculates the economy rate by dividing the total runs by the total overs, rounded to 2 decimal places.
* **SUM(4H) as 4W:** Sums the 4H column from the DATA CTE, representing the total number of times the bowler took 4 wickets in an inning.
* **SUM(5H) as 5W:** Sums the 5H column from the DATA CTE, representing the total number of times the bowler took 5 or more wickets in an inning.
* **SEASON:** Includes the SEASON column.
* The result is grouped by BOWLER and SEASON using the GROUP BY clause.
* The SQL query defined earlier is executed using spark.sql(query). The result is assigned to the DataFrame variable ans.
* **coalesce(1)** is used to repartition the DataFrame into a single partition. This is done to ensure that the data is written to a single output file.
* **dropna()** is called to remove any rows with null values from the DataFrame.
* A window specification named window\_spec is defined using **Window.partitionBy("Season").orderBy(desc("WICKETS")).** This specifies the partitioning of rows by the "Season" column and ordering within each partition based on the "WICKETS" column in descending order.
* ans is updated by adding a new column "POS" using withColumn("POS", **f.row\_number().over(window\_spec)).** This column assigns a row number starting from 1 within each partition, based on the ordering defined in the window specification.
* The desired columns for the final result are selected using ans.select(...). These columns include "POS", "BOWLER", "MATCHES", "INNINGS", "WICKETS", "RUNS", "OVERS", "ECON", "4W", "5W", and "SEASON".
* The final result is written to CSV format using ans.write.format("csv"). The option "header" is set to "true" to include column headers in the output file. The mode is set to "overwrite" to overwrite any existing data in the output path. The DataFrame is partitioned by the "Season" column using partitionBy("Season"). The output is saved to the specified output\_path with the file name "requriment\_4".

**REQUIREMENT 5:**

* The Requirement 1 is to calculate the “**Most Runs Conceded In Innings**”
* The ‘match file’ will contain the path of the file where the IPL\_Matches\_2008\_2022.csv is located
* The ‘ball file’ will contain the path of the file where the IPL\_Ball\_by\_Ball\_2008\_2022.csv is located
* **ETL STEPS:**
* match=spark.read.format("csv").option("header", "true").load(match\_file): This line reads the match data CSV file using the CSV format and sets the "header" option to true, indicating that the first row contains column names. The load(match\_file) part specifies the file path. The resulting DataFrame is assigned to the variable match.
* .withColumnRenamed("ID", "MATCH\_ID"): This part of the code renames the column "ID" in the match DataFrame to "MATCH\_ID" using the withColumnRenamed() function. The updated DataFrame is still assigned to the variable match.
* ball=spark.read.format("csv").option("header", "true").load(ball\_file): Similar to the previous line, this code reads the ball data CSV file using the CSV format and sets the "header" option to true. The resulting DataFrame is assigned to the variable ball.
* match\_ball=ball.join(f.broadcast(match), ball.ID==match.MATCH\_ID, "inner"): This line performs an inner join operation between the ball and match DataFrames. The join condition is specified as ball.ID==match.MATCH\_ID, which means the join is performed when the "ID" column in the ball DataFrame matches the "MATCH\_ID" column in the match DataFrame. The "inner" argument indicates that only matching records should be included in the result.
* SELECT statement Specifies the columns that will be selected in the result set.
* ROW\_NUMBER() OVER(...) AS POS: Assigns a row number to each row in the result set based on specific ordering criteria.
* BOWLER AS PLAYER: Retrieves the name of the bowler and aliases it as "PLAYER".
* COUNT(DISTINCT OVERS) AS OVERS: Calculates the count of distinct overs bowled by the player and aliases it as "OVERS".
* SUM(CASE WHEN EXTRA\_TYPE = 'NA' THEN CAST(TOTAL\_RUN AS INTEGER) ELSE 0 END) AS RUNS: Calculates the sum of runs conceded by the bowler, excluding extras, and aliases it as "RUNS".
* SUM(CASE WHEN ISWICKETDELIVERY = 1 THEN 1 ELSE 0 END) AS WICKETS: Calculates the total number of wickets taken by the bowler and aliases it as "WICKETS".
* CAST(COALESCE(SUM(CASE WHEN EXTRA\_TYPE = 'NA' THEN 1 ELSE 0 END) / SUM(CASE WHEN ISWICKETDELIVERY = 1 THEN 1 ELSE 0 END), 0) AS INTEGER) AS SR: Calculates the strike rate of the bowler by dividing the total number of runs conceded by the total number of wickets taken. The strike rate is then cast as an integer and aliased as "SR".
* CASE WHEN BattingTeam != TEAM1 THEN TEAM2 WHEN BattingTeam != TEAM2 THEN TEAM1 END AS Against: Determines the opposing team based on the "BattingTeam" column and aliases it as "Against". The condition checks if the "BattingTeam" is not equal to "TEAM1" and if so, returns "TEAM2". Similarly, if the "BattingTeam" is not equal to "TEAM2", it returns "TEAM1".
* VENUE: Retrieves the venue of the match.
* CAST(DATE AS DATE): Converts the "DATE" column to the DATE data type.
* SEASON: Retrieves the season of the match.
* FROM ALL: Specifies the table or view from which the data is retrieved. In this case, the table is named "ALL".
* GROUP BY: Groups the result set by specific columns.
* ID, BOWLER, BATTINGTEAM, TEAM1, TEAM2, VENUE, DATE, SEASON: Groups the result set by the specified columns.

**REQUIREMENT 6:**

* The Requirement 1 is to calculate the “**Fastest Fifties**”
* The ‘match file’ will contain the path of the file where the IPL\_Matches\_2008\_2022.csv is located
* The ‘ball file’ will contain the path of the file where the IPL\_Ball\_by\_Ball\_2008\_2022.csv is located
* **ETL STEPS:**
* glueContext.create\_dynamic\_frame.from\_options(): This method is used to create a dynamic DataFrame from various data source options.
* format\_options={"quoteChar": '"', "withHeader": True, "separator": ","}: The format\_options parameter is used to specify options related to the CSV file format. In this case, it sets the quote character to " (double quotes), enables the presence of a header row in the CSV file (withHeader: True), and specifies the separator as , (comma).
* connection\_type="s3": The connection\_type parameter specifies the type of connection to the data source. In this case, it is set to "s3" to indicate that the data is stored in an S3 bucket.
* format="csv": The format parameter specifies the format of the data source. Here, it is set to "csv" to indicate that the data is in CSV format.
* connection\_options={...}: The connection\_options parameter is used to provide additional options for connecting to the data source. In this case, it includes the following options:
* "paths": ["s3://groupno6/data/cricket\_databases/cricket\_csv/dataload=20230608/IPL\_Ball\_by\_Ball\_2008\_2022.csv"]: Specifies the S3 path(s) where the CSV file(s) are located. In this case, it provides a single path to the CSV file.
* "recurse": True: Indicates that the Glue should recursively search for files in subdirectories under the specified S3 path.
* transformation\_ctx="S3bucket\_node1": The transformation\_ctx parameter is used to assign a name to the transformation context. It is used to track and manage transformations in AWS Glue.
* By executing this code, a dynamic DataFrame named "S3bucket\_node1" will be created from the CSV file located in the specified S3 path. The DataFrame will have the CSV file's contents with the specified format options applied.
* Same as for S3bucket\_node2
* df1 = S3bucket\_node1.toDF(): and df2 = S3bucket\_node2.toDF(): These two converts the dynamic DataFrame into a static DataFrame . The toDF() method is used to perform this conversion.
* df\_merged = df1.join(df2, on="ID"): This line joins the two DataFrames (df1 and df2) based on the common column "ID". It performs an inner join by default, meaning only the matching records from both DataFrames will be included in the resulting DataFrame df\_merged.
* df\_merged = df\_merged.filter(df\_merged.extra\_type != 'wides'): This line filters the df\_merged DataFrame to exclude rows where the column "extra\_type" has the value "wides". The resulting DataFrame will only contain rows where the "extra\_type" is not "wides".
* df\_merged.createOrReplaceTempView("df\_merged"): This line creates a temporary view named "df\_merged" from the df\_merged DataFrame. A temporary view allows you to execute SQL queries against the DataFrame using its assigned name.
* df1.createOrReplaceTempView("ball"): This line creates a temporary view named "ball" from the df1 DataFrame. This view can be used to execute SQL queries against the df1 DataFrame using the "ball" view name.
* df2.createOrReplaceTempView("match"): Similarly, this line creates a temporary view named "match" from the df2 DataFrame. The "match" view can be used to execute SQL queries against the df2 DataFrame.
* The result\_df CTE selects all columns from df\_merged and adds two additional columns using window functions:
* result\_df\_ballnumber: It assigns a row number within each partition (defined by Season, Date, and batter) based on the batter column.
* cumulative\_runs: It calculates the cumulative sum of batsman\_run within each partition from the start of the partition to the current row.
* The cumulative\_df CTE selects all columns from result\_df and adds four additional columns based on certain conditions:
* 4s: It assigns a value of 1 if batsman\_run is equal to 4, otherwise 0.
* 6s: It assigns a value of 1 if batsman\_run is equal to 6, otherwise 0.
* BF: It assigns the value of result\_df\_ballnumber if cumulative\_runs is greater than or equal to 50, otherwise NULL.
* Against: It assigns the value of Team2 if BattingTeam is equal to Team1, otherwise Team1.
* The grouped\_df CTE selects specific columns from cumulative\_df and applies grouping and aggregation:
* It groups the data by Season, Date, batter, Against, and Venue.
* It calculates the minimum value of BF, the sum of 4s, 6s, and batsman\_run for each group.
* It applies a filter (HAVING) to exclude rows where BF is NULL.
* The ordered\_df CTE It selects specific columns from grouped\_df and applies a row number using the ROW\_NUMBER window function within each partition defined by Season, ordered by BF.
* Finally, the main query selects columns from ordered\_df and performs additional transformations:
* It assigns the row number (POS) from ordered\_df.
* It converts the Season column to an integer by mapping specific values to their corresponding years.
* It formats the Match Date column using the date\_format function.
* The resulting DataFrame is assigned to the variable out\_df.
* out\_df = out\_df.repartition("Season").coalesce(1): This line repartitions the out\_df DataFrame based on the "Season" column. It ensures that all rows with the same "Season" value are colocated within the same partition. The subsequent coalesce(1) operation reduces the number of partitions to 1, effectively creating a single output file.
* output\_dynamic\_frame = DynamicFrame.fromDF(out\_df, glueContext, "output\_dynamic\_frame"): This line converts the out\_df DataFrame into a dynamic frame named "output\_dynamic\_frame". The DynamicFrame.fromDF() method is used for this conversion, and it takes the DataFrame (out\_df), the GlueContext (glueContext), and a name for the dynamic frame as input.
* S3bucket\_node3 = glueContext.write\_dynamic\_frame.from\_options(...):
* This line writes the output\_dynamic\_frame dynamic frame to an S3 bucket.
* The frame parameter specifies the dynamic frame to be written (output\_dynamic\_frame).
* The connection\_type parameter is set to "s3" to indicate an S3 connection.
* The format parameter is set to "csv" to specify the output format as CSV.
* The connection\_options parameter is used to provide additional options for the S3 connection. It includes the following options:
* "path": "s3://groupno6/output/result6/": Specifies the S3 path where the output CSV files will be stored.
* "partitionKeys": ["Season"]: Indicates that the output should be partitioned based on the "Season" column.
* The transformation\_ctx parameter assigns a name to the transformation context, which is used to track and manage transformations in AWS Glue

**REQUIREMENT 7:**

* We Are Using DATAFRAMES in this requirement
* The s3bucket\_nodel will contain the address of ipl\_ball\_by\_ball\_2008\_2022\_csv
* The Amazons3\_nodel will contain the address of ipl\_matches\_2008\_2022\_csv
* The ApplyMapping\_node2 will contain all the columns present in ‘ipl\_ball\_by\_ball\_2008\_2022\_csv’
* The ChangeSchema\_node1 will contain all the columns present in ‘ipl\_matches\_2008\_2022\_csv’
* It reads the data from the two datasets (S3bucket\_node1 and AmazonS3\_node1686216371280) using the glueContext.create\_dynamic\_frame.from\_catalog function.
* **ETL STEPS:**
* The script applies mapping to the data using the ApplyMapping.apply function for both datasets, specifying the source and target columns.
* The script joins the two datasets based on the "id" column using the join method.
* It filters out rows where the "extra\_type" column is not equal to 'wides'.
* The script selects specific columns from the joined and filtered dataset to create a new DataFrame.
* It performs grouping and aggregation on the selected DataFrame to calculate various statistics such as total runs, boundaries (4s and 6s), and strike rate.
* The script renames and transforms certain columns in the DataFrame to make them more readable and meaningful.
* The DataFrame is ordered in descending order based on the number of boundaries (6s).
* It adds a new column for the position (POS) within each season using the row\_number function and windowing.
* The final DataFrame is converted back to a DynamicFrame using DynamicFrame.fromDF.
* Finally, the result is written to an S3 bucket in CSV format using the glueContext.write\_dynamic\_frame.from\_options function.