How to Extract the Quarter from a Date in PySpark

Authored by

stats writer

January 19, 2026

Introduction to Date Manipulation in PySpark

PySpark, the Python API for Apache Spark, is an indispensable tool for large-scale data processing and analysis. When dealing with time-series or transactional data, aggregating metrics based on specific time periods is frequently required. One of the most common requirements in financial and business intelligence reporting is the ability to segment data based on the fiscal or calendar quarter. This process, known as date part extraction, allows analysts to shift from row-level transaction data to period-over-period performance summaries. Successfully extracting the quarter from a date column within a PySpark DataFrame is fundamental for robust temporal analysis.

The core challenge in date processing is ensuring that the data type is correctly handled, and that the chosen extraction method is highly performant across a distributed cluster. PySpark's SQL functions module provides specialized tools designed to handle date and timestamp types efficiently. We will explore two primary methods that leverage built-in PySpark functions to extract the quarter information, addressing both the simple extraction of the quarter number and the more complex task of combining the year and quarter into a single, cohesive string identifier. These techniques are vital for creating standardized time dimensions in your analytical pipelines.

The following examples demonstrate how to achieve precise PySpark date manipulation. We will specifically focus on utilizing the powerful functions available in the `pyspark.sql.functions` library. Mastering these methods ensures that your data preparation steps are clean, performant, and easily replicable, forming the basis for advanced business logic and reporting requirements across massive datasets.

Method 1: Extracting Only the Quarter Number

The most direct approach to isolating the quarter relies on the highly optimized built-in `quarter` function found within `pyspark.sql.functions`. This function directly returns an integer value representing the quarter of the year (1 for January-March, 2 for April-June, etc.) for the provided date column. This method is typically used when the year context is already available in a separate column, or when the analysis strictly focuses on cyclical comparisons across different years using aggregation keys.

To implement this, we utilize the `withColumn` transformation available on the DataFrame. The `withColumn` function allows us to add a new column based on the application of a column expression. In this case, the expression is `quarter(df)`. This creates a new column, which we name 'quarter', populated by the integer result of the function applied to every row's 'date' value, providing a simple numeric classification.

Using this method keeps the transformation concise and highly readable, requiring only a single

import. The syntax presented below is the core logic necessary to perform the extraction.

from pyspark.sql.functions import quarter

df_new = df.withColumn('quarter', quarter(df))

Method 2: Combining Year and Quarter for Complete Context

While extracting only the quarter number is useful, analytical scenarios spanning multiple years often require a unique identifier that combines both the year and the quarter. This combined format, such as YYYYQQ (e.g., 2022Q1), serves as an excellent time key for data warehousing, pivot tables, and precise chronological visualization. To achieve this in PySpark, we must utilize a combination of four functions from `pyspark.sql.functions`: `year()`, `quarter()`, `concat()`, and `lit()`.

The process involves combining the numerical outputs of `year()` and `quarter()` with a fixed string separator ('Q'). The `concat` function is specifically designed to merge multiple column values or expressions into a single string column. It handles the implicit casting of the integer year and quarter results into strings before merging them sequentially.

The crucial element, the static letter 'Q', must be introduced using the `lit` (literal) function. The `lit` function converts a constant Python value into a column expression, which is mandatory for it to be accepted as an argument within the `concat` function. This ensures that the 'Q' separator is correctly applied across all rows of the DataFrame.

from pyspark.sql.functions import quarter, year, concat, lit

df_new = df.withColumn('year-quarter',
concat(year(df), lit('Q'), quarter(df)))

Setting Up the Demonstration DataFrame

To visualize these methods effectively, we must first initialize a SparkSession and create a sample DataFrame. This foundational setup allows us to execute the transformation code blocks and inspect the results. Our sample data includes two columns: `date`, which holds the date strings we intend to process, and `sales`, a numerical metric that would typically be aggregated during analysis.

The dates provided span across two different calendar years (2022 and 2023) and cover all four quarters of 2022, ensuring that our examples demonstrate the function's behavior across various

temporal segments. The ability of PySpark to handle date strings in standard formats like 'YYYY-MM-DD' without explicit parsing before applying functions like `quarter()` simplifies the initial data preparation workflow considerably.

The code below establishes the required environment and displays the starting point DataFrame, `df`.

from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data = ,
,
,
,
,
,
,
,
]

#define column names
columns =

#create dataframe using data and column names
df = spark.createDataFrame(data, columns)

#view dataframe
df.show()

+----------+-----+
| date|sales|
+----------+-----+
|2022-01-31| 6|
|2022-02-28| 8|
|2022-03-31| 10|
|2022-04-30| 5|
|2022-06-30| 4|
|2022-09-30| 8|
|2022-11-30| 8|

|2023-01-31| 3|
|2023-02-28| 5|
|2023-03-31| 14|
+----------+-----+

Execution of Example 1: Extracting Only the Quarter

Applying the transformation from Method 1 to our sample data demonstrates the speed and efficiency of the `quarter` function. The syntax is minimal, only requiring the import of the function and its application via `withColumn`. This results in the creation of a new column, `quarter`, which contains an integer from 1 to 4 corresponding to the month of the date record.

As shown in the output below, dates such as '2022-01-31', '2022-02-28', and '2022-03-31' are all correctly assigned the value 1, signifying the first quarter. Similarly, November 2022 is assigned 4, corresponding to Q4. This numerical categorization is extremely useful for subsequent data aggregation tasks where you might use standard Spark SQL grouping operations.

The new `quarter` column is appended to the original DataFrame, yielding `df_new`. This transformation is immutable in Spark, meaning the original `df` remains untouched, adhering to the standard functional programming paradigm used in PySpark.

from pyspark.sql.functions import quarter

#extract quarter from each string in 'date' column
df_new = df.withColumn('quarter', quarter(df))

#view new DataFrame
df_new.show()

+----------+-----+-------+
| date|sales|quarter|
+----------+-----+-------+
|2022-01-31| 6| 1|
|2022-02-28| 8| 1|
|2022-03-31| 10| 1|
|2022-04-30| 5| 2|
|2022-06-30| 4| 2|
|2022-09-30| 8| 3|
|2022-11-30| 8| 4|
|2023-01-31| 3| 1|
|2023-02-28| 5| 1|

|2023-03-31| 14| 1|
+----------+-----+-------+

The new **quarter** column contains the numerical quarter of each date in the **date** column, providing a clear basis for time segmentation.

Execution of Example 2: Combining Year and Quarter

This approach provides a composite temporal key that uniquely identifies the quarter within its specific year, solving the ambiguity inherent in multi-year data. By combining `year()`, `concat()`, `lit()`, and `quarter()`, we generate the descriptive 'YYYYQQ' format in a new column called `year-quarter`.

The resulting data showcases the distinction between different years clearly. Notice that '2022-03-31' is labeled '2022Q1', while '2023-03-31' is labeled '2023Q1'. This differentiation is essential for accurate time-series comparisons and prevents metrics from different years but the same quarter number from being incorrectly grouped together. This format is often the preferred output for dashboarding and reporting systems due to its immediate interpretability.

The efficiency of the Spark functions ensures that this multi-step concatenation operation is performed rapidly across the distributed data partitions. The **year-quarter** column, generated by the syntax below, is a powerful new dimension for complex analytical workflows.

from pyspark.sql.functions import quarter, year, concat, lit

#extract year and quarter from each string in 'date' column
df_new = df.withColumn('year-quarter',
concat(year(df), lit('Q'), quarter(df)))

#view new DataFrame
df_new.show()

+----------+-----+------------+
| date|sales|year-quarter|
+----------+-----+------------+
|2022-01-31| 6| 2022Q1|
|2022-02-28| 8| 2022Q1|
|2022-03-31| 10| 2022Q1|
|2022-04-30| 5| 2022Q2|
|2022-06-30| 4| 2022Q2|
|2022-09-30| 8| 2022Q3|

|2022-11-30| 8| 2022Q4|
|2023-01-31| 3| 2023Q1|
|2023-02-28| 5| 2023Q1|
|2023-03-31| 14| 2023Q1|
+----------+-----+------------+

The new **year-quarter** column contains the year and quarter of each date in the **date** column, providing a comprehensive temporal key. Note that we utilized the **concat** function to merge the numerical year and quarter with the 'Q' separator, which was included using the **lit** function to ensure it is treated as a column expression.

Deep Dive into Concat and Lit Functions

The functions `concat` and `lit` are foundational tools for generating complex string identifiers and metadata in Spark transformations. The `concat` function accepts an arbitrary number of Column objects and stitches their row values together. A key advantage in our date extraction example is its ability to automatically handle the conversion of the integer outputs from `year()` and `quarter()` into strings, eliminating the need for manual type casting functions like `cast('string')`. This simplifies the transformation logic significantly.

The `lit` function serves a crucial architectural purpose within PySpark. When defining a transformation using methods like `withColumn`, every argument used to calculate the new column's value must be a Column expression--either a reference to an existing column or the result of a Spark function applied to a column. Raw Python values (like the string 'Q') are not natively supported as transformation inputs. The `lit` function converts this static Python literal into a Column object that contains the same constant value across every row, making it compatible with the Spark execution engine.

Understanding the interplay between `concat` and `lit` is essential for constructing robust, high-performance data transformations that incorporate both dynamic (column-derived) and static (constant) data points, extending far beyond date manipulation to tasks such as creating standardized identifiers or flag columns across your entire DataFrame.