How to Extract the Year from a Date in PySpark

Authored by

stats writer

February 5, 2026

The ability to efficiently manipulate and segment date and time information is fundamental in large-scale data processing, particularly within Extract, Transform, Load (ETL) pipelines and analytical workflows. When working with PySpark, a powerful tool for distributed computing, extracting specific temporal components--such as the year--from a composite date field is a common requirement. This transformation allows analysts to group data by annual cohorts, analyze temporal trends, or filter records based on specific time boundaries. The process detailed here utilizes built-in DataFrame functions provided by the pyspark.sql.functions module, ensuring both high performance and concise code.

To successfully extract the year, the initial date column must be correctly interpreted by Spark, typically as a DateType or TimestampType. Once the data type is validated, we leverage the specialized `year` function, which is designed explicitly for this purpose. This function handles the underlying serialization and extraction logic within the distributed environment, ultimately returning the numerical year value as an integer type suitable for indexing or further numerical analysis. By mastering this simple, yet crucial, transformation step, users can significantly enhance their data aggregation and reporting capabilities within the PySpark ecosystem.

PySpark: Extract Year from Date

The Importance of Temporal Feature Engineering

In analytical data processing, converting a raw date column into discrete components--such as year, month, or day--is known as temporal feature engineering. This process is critical because many analytical models or reports require these segmented features rather than the complete date string or timestamp. For instance, determining year-over-year growth rates or segmenting customer behavior based on the year of registration necessitates isolating the year component. PySpark provides a suite of optimized functions to handle these transformations efficiently across massive datasets, leveraging the parallel processing capabilities of the underlying Apache Spark engine.

The `year` function is perhaps the simplest and most direct method for this specific extraction. It abstracts away the complexities of dealing with different date formats and time zones, provided the input column is recognized as a valid date type. Using native Spark functions like this ensures that the operation is executed optimally on the cluster workers, preventing unnecessary data shuffling or serialization issues that might arise from using User-Defined Functions (UDFs) written in pure Python. Therefore, understanding and utilizing the correct built-in functions is paramount for maintaining performance in large-scale data environments.

Core Syntax and Function Importation

To begin the extraction process in PySpark, the necessary function must first be imported from the `pyspark.sql.functions` module. This module serves as the central repository for most standard SQL-like operations and transformations applicable to Spark DataFrames. The specific function required is `year`. Once imported, this function is then applied within a column transformation operation, most commonly utilizing the `withColumn` method available on the DataFrame object. The combination of these tools allows for the creation of a new column derived directly from the existing date information.

The fundamental syntax involves importing the function and then defining the new column name, followed by the application of the `year` function to the target column. This approach is highly declarative, clearly stating the intended transformation. Observe the minimal required syntax below, which assumes an existing DataFrame named `df` that contains a date column:

from pyspark.sql.functions import year

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

This code block effectively creates a new column, named `year`, within the resulting DataFrame called `df_new`. The core functionality is driven by the `year(df)` expression, which targets the values in the `date` column and computes the corresponding year for each record. This transformation is executed lazily by Spark, meaning the computation only occurs when an action, such as `show()` or `write()`, is called on the resulting DataFrame.

Deep Dive into the PySpark Environment Setup

Before any transformation can take place, a Spark environment must be initialized. In PySpark, this begins with establishing a `SparkSession`, which acts as the entry point to all Spark functionality. The `SparkSession` allows the program to interact with the underlying cluster or local environment, managing resources and configurations necessary for distributed computation. Without a properly initialized session, operations such as defining data schemas or creating DataFrames cannot proceed.

For demonstration purposes, we will define a small dataset that simulates sales records, consisting of a date and the corresponding sales amount. This approach clearly illustrates how raw data is structured into a distributed DataFrame before the extraction operation is applied. It is crucial that the date strings in the input data conform to a standard format (like 'YYYY-MM-DD') so that Spark can correctly infer the schema or parse them into the necessary DateType during DataFrame creation.

Practical Example: Setting Up the Sales Data

The following example details the complete process, from initializing the Spark environment to creating the sample sales DataFrame. We define the data as a list of rows and specify the column names. The `SparkSession.builder.getOrCreate()` method ensures that we either retrieve an existing session or create a new one if none is running, providing a robust entry point for the subsequent steps.

We assume a scenario where a company tracks daily sales figures, and we need to analyze these sales segmented by the year in which they occurred. The structure of the data clearly displays the mixture of years (2021, 2022, and 2023) that the transformation function must successfully differentiate.

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|
+----------+-----+
|2021-04-11| 22|
|2021-04-15| 14|
|2021-04-17| 12|

|2022-05-21| 15|
|2022-05-23| 30|
|2023-10-26| 45|
|2023-10-28| 32|
|2023-10-29| 47|
+----------+-----+

The resulting DataFrame, `df`, now holds the raw temporal data in the `date` column. Our next objective is to perform the segmentation, isolating the year from this composite date field. This transformation is necessary if we intend to perform aggregation, such as calculating total sales per year, which is a common requirement in business intelligence and reporting.

Executing the Transformation and Reviewing Results

With the sample DataFrame successfully created, we can now proceed to apply the transformation using the imported `year` function and the `withColumn` method. The `withColumn` function is non-mutating; it returns a new DataFrame with the new column added, preserving the integrity of the original `df`. This functional programming approach is characteristic of Spark's design, favoring immutability for better distributed processing management.

The code below demonstrates the execution of the year extraction and then displays the resulting DataFrame, `df_new`. Notice how the extracted year, which is an integer, is appended as a distinct column, allowing for streamlined subsequent analysis. If the input column had been of an incompatible string format, this operation might have resulted in null values or an error, underscoring the importance of proper data type handling in PySpark.

from pyspark.sql.functions import year

#extract year from date column
df_new = df.withColumn('year', year(df))

#view new DataFrame
df_new.show()

+----------+-----+----+
| date|sales|year|
+----------+-----+----+
|2021-04-11| 22|2021|
|2021-04-15| 14|2021|
|2021-04-17| 12|2021|
|2022-05-21| 15|2022|

|2022-05-23| 30|2022|
|2023-10-26| 45|2023|
|2023-10-28| 32|2023|
|2023-10-29| 47|2023|
+----------+-----+----+

As evident in the output, the newly created **year** column successfully contains the four-digit year corresponding to each entry in the **date** column. This transformation is now complete, and the `df_new` DataFrame is ready for further processing, such as aggregation, filtering, or joining with other datasets based on this new temporal identifier. This concise methodology highlights the efficiency of leveraging native Spark SQL functions over slower, Python-based iterative methods.

Detailed Explanation of the `withColumn` Function

The `withColumn` function is perhaps the most essential tool for DataFrame transformation in PySpark. Its primary purpose is to add a new column or replace an existing one. When adding a new column, as done in our example, it takes two main arguments: the name of the new column (e.g., `'year'`) and the column expression that defines how the values are computed (e.g., `year(df)`). This expression can be any valid combination of Spark SQL functions applied to existing columns.

A significant characteristic of `withColumn` is its resilience and ease of use in chaining operations. Since it returns a new DataFrame, complex transformations involving multiple steps can be executed sequentially in a highly readable manner. Furthermore, if a column name passed to `withColumn` already exists, the function performs an update operation, overwriting the old content with the results of the new expression, which is useful for cleaning or recalculating features in place.

Alternative Methods for Date Component Extraction

While the `year` function is the most direct approach for extracting the year specifically, PySpark offers other powerful and flexible functions within the pyspark.sql.functions module that can also achieve similar results or offer more granular control. Understanding these alternatives is beneficial for scenarios where multiple date components need extraction simultaneously or when highly customized formatting is required.

One primary alternative is the `date_format` function. This function allows users to specify a format string (e.g., 'yyyy' for the year) to extract or reformat a date component. While slightly more verbose for a simple year extraction, `date_format` is indispensable when you need output in a specific string representation, such as converting the year to a two-digit format ('yy') or combining

date components into a custom string. Another powerful function is `extract`, which uses ANSI SQL syntax for extracting specific fields like YEAR, MONTH, or DAY from a date or timestamp column, providing a SQL-standard alternative to the specialized functions.

Summary of Best Practices in Date Manipulation

Successfully manipulating temporal data in PySpark relies on adhering to several best practices. Firstly, always ensure that the input column is cast to a native Spark DateType or TimestampType before applying functions like `year`. While Spark is often intelligent enough to handle common string formats, explicit casting removes ambiguity and prevents performance degradation caused by repeated parsing. Secondly, prioritize built-in functions from pyspark.sql.functions over Python UDFs, as the former are highly optimized for cluster execution.

The methodology demonstrated--importing specific functions and applying them using the `withColumn` paradigm--represents the gold standard for DataFrame transformations. By consistently applying these principles, data engineers can ensure their ETL and data preparation scripts are scalable, maintainable, and highly efficient, regardless of the volume of data being processed. The extraction of the year is a foundational skill that opens the door to much more complex and rewarding temporal analysis within the Spark environment.

The new **year** column contains the year of each date in the **date** column, confirming the success of the transformation.

Note that we utilized the **withColumn** function to seamlessly add a new column called **year** to the DataFrame while ensuring all existing columns and their data remained intact.

Note: You can find the complete documentation for the PySpark **withColumn** function online.

Further Exploration in PySpark Date Manipulation

Mastering the extraction of date components is just the first step in comprehensive temporal data handling. PySpark provides numerous tutorials explaining how to perform other common tasks, enabling full control over date and time arithmetic, comparisons, and formatting:

Calculating the difference between two dates in days or months.
Adding or subtracting intervals (e.g., adding 30 days to every date).
Converting local timestamps to UTC or vice versa, handling time zone differences.
Extracting specific details like the day of the week or the quarter of the year.

By exploring these advanced features, users can handle complex business logic and build robust, time-sensitive analytical features for their data models. The core principle remains the same: leverage the optimized functions provided by the `pyspark.sql.functions` module for maximum

performance and clarity.