How to Extract the Year from a Date in PySpark

Authored by

stats writer

January 19, 2026

How to extract the year from a date is a fundamental operation in PySpark data analysis, particularly when working with large datasets where time series aggregation is required. The DataFrame structure, central to PySpark operations, necessitates the use of specific, optimized built-in functions for manipulating date and time objects efficiently across distributed clusters. This process is crucial for tasks ranging from monthly sales aggregation to calculating annual growth rates, ensuring that the temporal components of your data are accurately segmented before further processing.

We achieve this extraction primarily by leveraging the powerful functions available within the PySpark SQL module, designed for high-performance transformations. The approach generally involves two key stages: first, ensuring the column is recognized as a proper date type, often handled using the to_date() function if the data is initially stored as a string; and second, applying the specialized year() function to isolate the year component. Unlike standard Python datetime operations which can be computationally slow on large volumes, PySpark's methodology is highly optimized for performance in big data environments. We will detail this process, showing exactly how to create a new column containing only the year value, making subsequent analytical operations straightforward and resource-efficient.

Setting Up the PySpark Environment

Before diving into the extraction process, it is paramount to establish an active **SparkSession**. The SparkSession acts as the entry point to utilizing PySpark functionality, managing resources, and allowing interaction with the underlying Apache Spark cluster. Without a properly initialized SparkSession, you cannot create or manipulate DataFrame objects, as the session handles the necessary configuration and resource allocation across the distributed environment. This setup routine is standard practice across all PySpark workflows, guaranteeing that the distributed computing engine is ready to handle the data transformation tasks requested efficiently.

In most interactive environments like Jupyter notebooks or standard Python scripts, the initialization process involves importing the necessary modules and then utilizing the builder pattern to either retrieve an existing session or create a new one. The `getOrCreate()` method is highly recommended when defining the session, as it intelligently prevents the creation of redundant Spark instances, thereby optimizing resource utilization and speeding up development cycles. Once the session is active, we gain immediate access to the extensive library of SQL functions, including all the specialized tools designed for high-performance date and time manipulation.

This initial step ensures that any subsequent operations, such as loading data into a DataFrame or applying complex transformations like date extraction, are executed with the full power of the distributed Spark architecture. Properly setting up the environment is the foundation for reliable

and scalable data processing in PySpark.

Core PySpark Functions for Date Extraction

The extraction of the year relies fundamentally on functions imported from `pyspark.sql.functions`. While several functions exist for handling various aspects of dates, the most direct and efficient method for isolating the year number is the use of the year() function. This function takes a column of date or timestamp type as its input and returns the integer representation of the year for every record in that column. It is a highly specialized function that simplifies complex date formatting logic into a single, declarative operation, removing the need for verbose user-defined functions (UDFs) which can sometimes hinder performance.

It is critical, however, to ensure that the source column is recognized by PySpark as a **DateType** or **TimestampType**. If the data is ingested from a raw source (like a CSV or JSON file) where the date is stored as a string (e.g., 'YYYY-MM-DD'), explicit conversion must occur first. This is where the to_date() function becomes invaluable. This function parses a string column based on a specified format and converts it into a proper SQL Date type, which is a necessary prerequisite for the successful application of the `year()` function. If the column is already correctly typed, this conversion step can be omitted, further streamlining the processing code.

Another crucial function involved in the overall workflow is withColumn(). This DataFrame transformation function allows us to add a new column derived from an existing one, or modify an existing column, without altering the underlying immutable structure of the DataFrame except for the addition of the new field. When extracting the year, we typically employ `withColumn()` to apply the `year()` function and store the resulting year integer in a newly named column, such as 'extracted_year' or simply 'year'.

The Fundamental Syntax for Year Extraction

The most straightforward and recommended approach for extracting the year involves importing the necessary function and applying it directly to the target DataFrame column using the withColumn() function. This method is highly concise, exceptionally readable, and adheres strictly to the functional programming paradigm favored in Spark operations, promoting clarity and maintainability. It is a robust method, provided the input column is of a compatible date or timestamp type; otherwise, conversion using a function like `to_date()` must precede the extraction.

The following syntax snippet illustrates the fundamental operation required to add the extracted year as a new field. We begin by importing `year` from the SQL functions library to make it available in our script. Subsequently, we call `withColumn` on our existing DataFrame (referenced here as `df`). The first argument specifies the name of the new output column (designated as 'year' in this

example), and the second argument applies the imported year() function directly to the existing 'date' column within the DataFrame. This declaration creates a derived column efficiently across the distributed cluster, ensuring scalability.

You can use the following standard syntax to extract the year from a date column within a PySpark DataFrame:

```
from pyspark.sql.functions import year

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

This powerful and common example creates a new column called **year** that efficiently extracts the integer year component from the date values contained in the **date** column of the original DataFrame. This transformation is the foundation upon which more complex temporal analyses and aggregations are subsequently built in PySpark.

Practical Implementation: Defining the Sample Data

To demonstrate the year extraction process effectively and provide a clear, executable example, we will work with a sample DataFrame representing sales records over several years. This scenario is highly representative of common use cases in real-world data science projects, where raw transactional data needs to be structured temporally for analysis. Our initial task is to initialize a SparkSession and populate a DataFrame with synthetic data containing a 'date' column and a corresponding 'sales' metric.

We utilize the `SparkSession.builder.getOrCreate()` method to initialize the environment, ensuring Spark is available and ready for processing. The data itself is defined as a list of lists, where the first element is the date string (in the standard 'YYYY-MM-DD' format) and the second element is the corresponding sales quantity. Defining the column names explicitly as 'date' and 'sales' allows us to use `spark.createDataFrame` to instantiate the DataFrame object successfully, maintaining control over the initial data schema.

The code block below illustrates the concise steps required for creating and initially displaying the sample DataFrame. While the 'date' column appears highly structured, its data type depends on Spark's schema inference during creation. Although the `year()` function is robust enough to often handle common date string formats automatically, relying on explicit type casting via to_date() function remains the safest practice for handling production data diversity. For this educational example, we proceed directly to the extraction assuming standard format recognition.

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

```
# Define sample data for sales transactions across multiple years
data = ,
,
,
,
,
,
]

# Define the column names for the DataFrame
columns =

# Create the DataFrame using the data and defined columns
df = spark.createDataFrame(data, columns)

# Display the initial DataFrame structure and content
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|
+----------+-----+
```


Executing the Year Extraction Operation

Our objective is now to take the `df` we created and derive a new column containing only the integer year value corresponding to each entry in the 'date' column. This transformation is pivotal for tasks like grouping sales data by year to identify long-term trends or preparing features for advanced time-series models that rely on clean, numerical temporal components. We will use the crucial combination of the year() function and the withColumn() function to achieve this modification efficiently and scalably.

First, we ensure the `year` function is imported from `pyspark.sql.functions`. We then chain the `withColumn()` transformation directly to the original DataFrame (`df`). We specify the new column name as 'year'. Critically, inside `withColumn()`, we apply the `year()` function, pointing it specifically to `df`. Because of Spark's architecture, this operation is executed in parallel across the cluster partitions, resulting in high throughput even for petabyte-scale datasets. The resulting DataFrame, `df_new`, maintains all the original columns while including the newly calculated 'year' column.

Viewing the resulting DataFrame (`df_new`) confirms that the extraction was successful and instantaneous. The newly generated 'year' column accurately reflects the year component of the corresponding dates (2021, 2022, and 2023). This derived column is now immediately ready to be used in powerful aggregation functions (such as `df_new.groupBy('year').sum('sales')`) or precise filtering operations, drastically simplifying subsequent analysis steps that rely on annual segmentation and reporting.

from pyspark.sql.functions import year

# Extract year from the existing 'date' column and store it in a new column named 'year'
df_new = df.withColumn('year', year(df))

# View the resulting DataFrame with the newly added 'year' column
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 shown in the output above, the new **year** column successfully contains the integer representation of the year for each corresponding date in the **date** column, confirming the robust functionality of the year() function combined with the immutable transformation capabilities of the DataFrame API.

Deep Dive into withColumn() for DataFrame Transformations

The withColumn() function is arguably one of the most fundamental and frequently used methods for PySpark DataFrame manipulation. Its primary role is to enable the modification or creation of columns based on arbitrary expressions defined by the user. When used for date extraction, as we have demonstrated, it ensures that the original DataFrame remains immutable--a core principle of Spark and functional programming--by generating a new DataFrame object that incorporates the calculated column.

The function signature requires two primary parameters: the name of the column to be added or replaced, and the Column expression that defines the logic for calculating the new values. This expression is precisely where we embed our specific SQL functions, such as `year()`, `to_date()`, or even complex conditional statements involving multiple columns. Because Spark utilizes lazy evaluation, the expression defined within `withColumn()` is not actually computed until an action (like `show()`, `count()`, or `write()`) is called, allowing the Catalyst optimizer to analyze and potentially rearrange the execution plan for maximum efficiency.

While one could technically achieve a similar result using the `select()` function by aliasing the expression, `withColumn()` is generally the preferred method when the intent is to retain all existing columns while adding a new, derived column. For instance, if you were only interested in the date and the newly extracted year, `select(df, year(df).alias('year'))` would work. However, for complex, iterative transformation workflows where preserving the full context of the original data is crucial, `withColumn()` provides a cleaner, more readable, and semantically appropriate approach, making the code easier to maintain and debug.

Note that we used the **withColumn** function to add a new column called **year** to the DataFrame while keeping all existing columns the same.

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

Handling Complex Date Formats using to_date()

In real-world data pipelines, dates are rarely delivered in a perfectly clean, ISO standard format ('YYYY-MM-DD'). Sometimes they arrive as arbitrary strings like 'DD/MM/YYYY', 'MM-DD-YY', or even include extraneous time zone information within a non-standard string structure. If PySpark cannot automatically infer the schema or if the format is non-standard, applying the year() function directly will typically result in null values or execution errors because the underlying type is not a valid DateType. This situation dictates that an explicit data type conversion step must be integrated into the workflow.

The to_date() function is the essential solution for this common data quality challenge. It requires

two mandatory arguments: the column containing the date strings and the format pattern string that accurately corresponds to the structure of the input data. For example, if your raw data column contained '26/10/2023', the necessary format pattern string would be 'dd/MM/yyyy'. This conversion function is then nested within our `withColumn` transformation, ensuring the data type is corrected before the specialized functions like `year()` are applied.

If our DataFrame contained a string column named 'raw_date' in the format 'MM/dd/yyyy', the comprehensive, multi-step extraction syntax would look like the following code example. This approach emphasizes safety and correctness by first casting the potentially ambiguous string data into a reliable DateType using `to_date()`, and then performing the extraction using `year()`, ensuring maximum stability when dealing with diverse and variable data sources.

from pyspark.sql.functions import year, to_date

df_converted = df.withColumn('date_converted', to_date(df, 'MM/dd/yyyy'))
df_final = df_converted.withColumn('extracted_year', year(df_converted))

Related PySpark Data Transformation Tutorials

The following tutorials explain how to perform other common data transformation and manipulation tasks in **PySpark**, building upon the foundational knowledge of column modification and function application demonstrated in this guide:

Extracting the month, day, or quarter from a date column using similar built-in functions.

Calculating the difference in days or months between two date columns using `datediff()`.

Converting PySpark DataFrames to Pandas for localized analysis on smaller, filtered subsets.

Using `date_format()` for customized date string outputs that include year, month, and day in specific formats.

Working with Timestamp data types and handling time zone conversions effectively.