How to Extract Minutes from a Timestamp in PySpark

Authored by

stats writer

February 4, 2026

Working with time series data is a fundamental requirement in modern data engineering and analytical workflows. When processing large datasets using Apache PySpark, precise manipulation of date and time components is essential for tasks ranging from feature engineering to aggregation. One common requirement is the need to isolate specific components, such as the minute value, from a complete timestamp column. The process of extracting the minutes from a timestamp in PySpark involves leveraging the powerful built-in functions and methods provided by the PySpark library. These functions allow for the efficient handling and analysis of time-related data within PySpark by enabling the conversion of a timestamp into an object that can be manipulated to extract specific components such as minutes.

Understanding Timestamp Handling in PySpark

When working within the PySpark SQL context, all time-based operations are managed primarily through the `pyspark.sql.functions` module, often imported simply as `F`. This approach ensures that operations are executed efficiently using the Catalyst Optimizer and are highly scalable, which is essential for processing big data volumes. Extracting specific time components, such as the minute, is a straightforward task facilitated by the rich set of functions available in this module.

We will examine two distinct but equally useful techniques for managing the minute component of a timestamp within a distributed DataFrame structure. The choice between these methods depends entirely on the desired output format--whether you require a discrete integer representing the minute (0-59) or a full timestamp object adjusted to the start of that minute.

The following sections detail both methods, providing the necessary syntax and practical examples. It is vital to ensure that your column containing the time information is correctly cast as a timestamp data type before attempting these extraction methods, as they rely on the underlying time structure for accurate parsing.

Method 1: Extracting the Numerical Minute Value (F.minute)

The first and perhaps most direct method involves utilizing the built-in `F.minute()` function, which is sourced from the pyspark.sql.functions module. This function is designed specifically to return the minute component as an integer, ranging from 0 to 59, irrespective of the other time components like hour, second, or date. This method is frequently employed when conducting frequency analysis, creating temporal bins, or generating features for machine learning models that require discrete time indicators.

To implement this, you typically use the `withColumn` transformation on your existing DataFrame. The `withColumn` operation creates a new column--in this case, perhaps named `minutes_extracted`--which holds the minute value derived by applying `F.minute()` directly to the source timestamp column. This operation is non-mutating with respect to the original column,

preserving the integrity of the raw time data for subsequent analyses.

The syntax for this approach is concise and highly readable. The following code block demonstrates how to apply this function to a column named `ts` to retrieve the specific minute of the hour:

```
from pyspark.sql import functions as F

df_new = df.withColumn('minutes', F.minute(df))
```

If the input timestamp value is **2023-01-15 04:14:22**, then applying this syntax would accurately return the integer value **14**. This approach offers maximum specificity when only the numerical minute component is required for downstream processing, such as calculating average latency per minute slot.

Method 2: Truncating the Timestamp to the Minute Level (F.date_trunc)

The second essential method involves using the `F.date_trunc()` function. This function serves a fundamentally different purpose than `F.minute()`; instead of returning an integer, `F.date_trunc()` returns a full timestamp value. This new timestamp is normalized, or truncated, to the beginning of the specified time unit. When used with the argument `'minute'`, all subsequent time components, namely seconds and milliseconds, are reset to zero.

This truncation method is invaluable when performing time-based aggregations, such as counting events or calculating averages per minute interval. By aligning all timestamps that fall within the same 60-second window to an identical reference point (the start of that minute), we greatly simplify the grouping process, ensuring accurate and consistent aggregation boundaries across a large dataset.

The `F.date_trunc()` function requires two positional parameters: the string literal specifying the format unit (e.g., `'minute'`, `'hour'`, `'day'`), and the column containing the source timestamp. This function is highly versatile, making it a foundational tool for complex time series data preparation in PySpark.

The implementation below illustrates how to apply `F.date_trunc()` to the timestamp column. Note that the output is still a timestamp object, making it suitable for subsequent temporal joins or time-window calculations.

```
from pyspark.sql import functions as F

df_new = df.withColumn('minutes', F.date_trunc('minute', df))
```

For an input timestamp of **2023-01-15 04:14:22**, using the `F.date_trunc('minute', ...)` function yields the output timestamp **2023-01-15 04:14:00**. This process discards the specific second (22) and normalizes the time data to the boundary of the 14th minute.

Setting Up the PySpark Environment and Sample Data

To demonstrate these methods practically, we must first initialize a Spark session and construct a representative sample DataFrame containing temporal data. Our sample scenario involves tracking sales transactions recorded at various timestamps, which simulates a common data engineering task.

Initializing the `SparkSession` is the mandatory first step for accessing all PySpark functionalities. Following the session creation, we define our sample data, which consists of transaction timestamps (stored initially as strings) and corresponding sales figures. Crucially, before any time extraction or manipulation can occur, the string representation of the timestamps must be explicitly converted to the PySpark `TimestampType` using the `F.to_timestamp()` function. This conversion step is critical, as specialized time functions will fail if the input data type is not a valid timestamp.

The following detailed setup code demonstrates the entire initialization sequence, from creating the Spark session to casting the data type and displaying the resulting DataFrame, which is now correctly formatted and ready for the extraction examples:

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

from pyspark.sql import functions as F

#define data
data = ,
,
,
]

#define column names
columns =

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

#convert string column to timestamp
df = df.withColumn('ts', F.to_timestamp('ts', 'yyyy-MM-dd HH:mm:ss'))

```
#view dataframe
df.show()

+-------------------+-----+
| ts|sales|
+-------------------+-----+
|2023-01-15 04:14:22| 225|
|2023-02-24 10:55:01| 260|
|2023-07-14 18:34:59| 413|
|2023-10-30 22:20:05| 368|
+-------------------+-----+
```

As shown in the output, the `ts` column is now correctly parsed as a timestamp, containing both the date and high-precision time details. This DataFrame, `df`, is the required input for applying our minute extraction techniques.

Illustrative Code Example for F.minute

We can use the following syntax to extract only the numerical minute value from each timestamp in the **ts** column of the DataFrame. This method is crucial when the goal is to create a feature that reflects the time slot within the hour, often used for identifying cyclical patterns in transaction volume.

We apply the `F.minute()` function to the `ts` column and assign the result to a new column named `minutes`. This transformation is executed lazily by PySpark and results in an integer column containing values from 0 to 59. This result is optimized for numerical aggregation or comparison operations.

The following code block demonstrates the extraction process and displays the resulting DataFrame, clearly illustrating how the function isolates the minute value from the full timestamp:

```
from pyspark.sql import functions as F

#extract minutes from each timestamp in 'ts' column
df_new = df.withColumn('minutes', F.minute(df))

#view new DataFrame
df_new.show()

+-------------------+-----+-------+
| ts|sales|minutes|
+-------------------+-----+-------+
```

|2023-01-15 04:14:22| 225| 14|
|2023-02-24 10:55:01| 260| 55|
|2023-07-14 18:34:59| 413| 34|
|2023-10-30 22:20:05| 368| 20|
+-------------------+-----+-------+

The new `minutes` column successfully shows only the integer minute value extracted from each timestamp in the **ts** column. For instance, the transaction that occurred at `04:14:22` correctly yields `14` in the dedicated minutes column.

Illustrative Code Example for F.date_trunc

This example showcases the application of `F.date_trunc()` to return each timestamp from the **ts** column truncated precisely to the minute boundary. This methodology is necessary when preparing data for time-windowed aggregations where the seconds component must be standardized.

We utilize the `withColumn` operation again, applying `F.date_trunc` with the string unit `'minute'`. It is important to reiterate that the output data type of the new column will be a timestamp, making it suitable for subsequent temporal grouping keys.

The resulting column aligns all timestamps within the same minute interval to an identical start time, simplifying grouping logic dramatically. This operation is highly efficient due to Spark's native function optimization.

from pyspark.sql import functions as F

#create new column that contains timestamp truncated to the minutes
df_new = df.withColumn('minutes_truncated', F.date_trunc('minute', df))

#view new DataFrame
df_new.show()

+-------------------+-----+-------------------+
| ts|sales|minutes_truncated|
+-------------------+-----+-------------------+
|2023-01-15 04:14:22| 225|2023-01-15 04:14:00|
|2023-02-24 10:55:01| 260|2023-02-24 10:55:00|
|2023-07-14 18:34:59| 413|2023-07-14 18:34:00|
|2023-10-30 22:20:05| 368|2023-10-30 22:20:00|
+-------------------+-----+-------------------+

The new `minutes_truncated` column clearly shows that all seconds information has been successfully reset to `00`, demonstrating the effective alignment of the timestamp to the start of the minute interval.

Advanced Considerations: Time Zone Awareness

A critical technical aspect when dealing with timestamps in PySpark is time zone management. Spark typically operates by storing timestamps internally as Coordinated Universal Time (UTC). However, the output of functions like `F.minute()` and `F.date_trunc()` will reflect the time zone configured for the Spark Session, which often defaults to the local system time zone.

If the source data originates from systems operating in different time zones or if daylight saving time (DST) transitions are involved, inconsistencies in minute extraction can arise. For instance, if an operation is performed exactly at a DST change, the interpretation of the hour, and thus the minute within that hour, might be skewed if the time zone is not standardized beforehand.

For maximum data integrity and reproducibility, especially in distributed environments, it is best practice to explicitly convert all timestamps to a known standard zone, such as UTC, using `F.to_utc_timestamp()` immediately after the initial string-to-timestamp conversion. This standardization prevents ambiguity and ensures that both extraction and truncation operations are performed on a consistent temporal reference point across the cluster.

Summary of Time Manipulation Techniques

The ability to accurately and efficiently handle temporal data is indispensable in large-scale data processing environments utilizing PySpark. By mastering the distinction between the `F.minute()` function for extracting integer components and the `F.date_trunc('minute', ...)` function for standardizing timestamps to the minute boundary, data engineers can perform powerful feature engineering and precise temporal aggregations.

Both methods demonstrate the simplicity and power embedded within the `pyspark.sql.functions` library. Selecting the correct function is a critical design choice dependent on the immediate analytical goal: isolation (integer extraction for feature building) versus normalization (timestamp truncation for aggregation grouping). Always confirm that the input column is correctly cast as a timestamp type to prevent common data type mismatch errors.

Beyond minute extraction, PySpark offers extensive capabilities for temporal manipulation. These include calculating time differences, extracting other components (like day of week or quarter), and implementing sophisticated sliding window aggregations. Continued exploration of the built-in functions ensures maximum utilization of Spark's optimized features for all time series analysis needs.

The following tutorials explain how to perform other common tasks in PySpark:

Extracting the day of the week or month from a date field.

Using `F.datediff()` to calculate the number of days between two columns.

Applying window functions for temporal rolling averages.