How to Get Minutes from a Timestamp in PySpark: A Step-by-Step Guide

Authored by

stats writer

January 2, 2026

Handling date and time data is a fundamental requirement in large-scale data processing, and PySpark provides robust, highly optimized SQL functions to manage this task efficiently across distributed clusters. Extracting specific components, such as the minute value, from a detailed timestamp column is a common data preparation step required for aggregation or feature engineering.

While PySpark offers functions like `from_unixtime()` for converting timestamps stored as a Unix epoch (a long integer) into a formatted string, the most direct and idiomatic way to extract the numerical minute component is by leveraging specialized date/time functions available within the `pyspark.sql.functions` module. These functions simplify the process significantly, allowing developers to focus on analysis rather than complex string manipulation.

This guide explores two primary, efficient methods for working with timestamp precision in PySpark: first, isolating the minutes as an integer value, and second, truncating the entire timestamp down to the minute level for grouping purposes. Understanding the difference between these approaches is crucial for accurate time-series data analysis.

Core Methods for Timestamp Manipulation in PySpark

When working with time-series data in PySpark, you often need to adjust the granularity of your timestamps. The choice of method depends entirely on your desired output: do you need a simple integer representing the minute (e.g., '14'), or do you need a new timestamp object where seconds and milliseconds are zeroed out (e.g., '2023-01-15 04:14:00')?

Both methods rely heavily on importing the required functionality from `pyspark.sql.functions`, conventionally aliased as `F`. This practice enhances code readability and minimizes namespace conflicts when working with diverse SQL operations. Below, we introduce the primary functions used to achieve minute extraction and truncation.

The following methods are highly recommended because they operate directly on the native Spark Timestamp data type, ensuring optimal performance across the distributed cluster compared to complex user-defined functions (UDFs) or string conversions.

Method 1: Extracting Numerical Minutes using F.minute()

This method is the most straightforward way to retrieve the specific minute value from a timestamp column. The F.minute() function takes a column expression as input and returns the minute component as an integer (ranging from 0 to 59). This result is highly useful when aggregating data based on specific minutes within an hour, or when creating time-based features for machine learning models.

To implement this, you simply use the `withColumn` transformation on your DataFrame, defining a new column that applies the `F.minute()` function to your existing timestamp column. This transformation is lazy and executed efficiently by the Spark engine.

Consider the following syntax snippet demonstrating how to apply `F.minute()`:

```
from pyspark.sql import functions as F

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

If the original timestamp is **2023-01-15 04:14:22**, then executing this syntax would return the integer value **14** in the newly created column. Note that the output is a numerical representation, discarding all date, hour, and second information.

Method 2: Truncating Timestamp Precision using F.date_trunc()

In contrast to extraction, truncation involves resetting the precision of the existing timestamp while maintaining the overall date and time structure. The F.date_trunc() function is invaluable for this purpose. It allows you to specify a unit (like 'minute', 'hour', 'day', 'month', etc.), and it returns a new timestamp where all components smaller than that unit are set to their minimum value (e.g., seconds become 00).

When truncating to the 'minute' unit, all seconds and sub-second components of the original timestamp are set to zero. This technique is indispensable for generating time windows or ensuring that multiple events occurring within the same minute are grouped together for aggregation tasks, such as calculating traffic flow per minute.

The syntax requires specifying the desired unit as the first argument, followed by the column expression:

```
from pyspark.sql import functions as F

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

If the original timestamp was **2023-01-15 04:14:22**, this syntax would return the full timestamp **2023-01-15 04:14:00**. It is important to remember that the output remains a Timestamp data type, unlike Method 1, which returns an integer.

Setting Up the PySpark Environment and Sample Data

To demonstrate these two methods practically, we first need to initialize a PySpark session and

create a sample DataFrame containing time-series data. Our hypothetical dataset tracks sales records, each associated with a specific timestamp.

The initial timestamps are provided as strings, which is common in raw datasets. Before any time-based transformations can occur, we must convert this string column into a proper Spark Timestamp data type using the `F.to_timestamp()` function, specifying the exact format string (`'yyyy-MM-dd HH:mm:ss'`) to ensure correct parsing. This conversion step is critical for allowing native Spark date/time functions to operate efficiently.

The following example shows how to set up the necessary components, create the DataFrame, and perform the initial type casting:

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

from pyspark.sql import functions as F

# Define sample data structure representing sales events
data = ,
,
,
]

Define column schema
columns =

# Create DataFrame
df = spark.createDataFrame(data, columns)

# Convert string column 'ts' to the appropriate Timestamp type
df = df.withColumn('ts', F.to_timestamp('ts', 'yyyy-MM-dd HH:mm:ss'))

# View the resulting DataFrame structure
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|

```
+-------------------+-----+
```

Practical Example 1: Isolating the Numerical Minute Value

Using the prepared DataFrame from the previous step, we can now apply Method 1 to isolate the specific minute number (0-59) from the `ts` column. This is achieved using the F.minute() function, which is designed precisely for this scalar extraction task.

We create a new column, creatively named `minutes`, that holds this extracted integer. This new column can then be used in subsequent operations, such as grouping sales data to see which minute within the hour is generally the busiest, independent of the actual date or hour of the transaction.

Executing the transformation is straightforward, demonstrating the conciseness of PySpark's built-in SQL functions:

```
from pyspark.sql import functions as F

# Apply F.minute() to extract the integer minute value
df_new = df.withColumn('minutes', F.minute(df))

# View the DataFrame with the new extracted minute column
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|
+-------------------+-----+-------+
```

As visible in the output, the new **minutes** column successfully shows only the numerical minute value derived from each corresponding timestamp in the `ts` column, effectively demonstrating the power and simplicity of the F.minute() function.

Practical Example 2: Truncating the Timestamp to Minute Precision

For scenarios where you need to maintain the full context of the timestamp (date, hour, and minute) while ignoring sub-minute precision (seconds and milliseconds), Method 2 using

F.date_trunc() is the appropriate solution. This function ensures that every record within the same 60-second window is mapped to an identical timestamp value, facilitating accurate time window grouping.

By passing the string literal `'minute'` as the unit, we instruct Spark to zero out the seconds component. This is especially useful for ETL pipelines where standardization of time measurement is required before joining datasets or calculating rate changes.

Observe the output when applying `F.date_trunc('minute', ...)` to the same sample data:

from pyspark.sql import functions as F

# Apply F.date_trunc() to standardize the timestamp precision
df_new = df.withColumn('minutes_truncated', F.date_trunc('minute', df))

# View the DataFrame with the new truncated timestamp column
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 column, **minutes_truncated**, clearly demonstrates how the original timestamp values have been adjusted. For instance, the first entry, originally ending in `:22` seconds, now ends in `:00` seconds, confirming that the precision was successfully truncated to the minute level.

Summary of Differences and Use Cases

The core difference between these two methods lies in the data type and scope of the output. Choosing the correct function is vital for maintaining data integrity and achieving the intended analytical outcome.

We can summarize the key differences:

F.minute(): Extracts the minute component as an **Integer** (0-59). Use this when you need a numerical value for counting or feature creation, where the rest of the date/time information is irrelevant.

F.date_trunc('minute', ...): Returns a new **Timestamp** object. Use this when you need to group records into time bins or standardize time series, maintaining the date, hour, and minute context while discarding seconds.

By mastering both F.minute() and F.date_trunc(), data engineers can effectively manipulate time data in PySpark to meet virtually any requirement for time-series aggregation or analysis.

Further Exploration of PySpark Time Functions

Beyond minutes, PySpark offers a rich set of built-in functions for extracting all temporal components, including `F.hour()`, `F.dayofmonth()`, `F.weekofyear()`, and `F.year()`. These functions follow the same straightforward syntax as F.minute(), making it easy to create granular features from raw timestamp data.

Furthermore, if your goal is to calculate the difference between two timestamps or shift time windows, explore functions like `F.datediff()` or `F.add_months()`. Utilizing these vectorized SQL functions is essential for writing scalable and performant code in a distributed environment.

We recommend exploring the official PySpark documentation for a complete list of date and time manipulation functions to enhance your data transformation pipelines.