How to Extract the Hour from a Timestamp in PySpark

Authored by

stats writer

February 4, 2026

Working with temporal data is a foundational requirement in most large-scale data processing pipelines. When utilizing PySpark--the Python API for Apache Spark--efficiently handling date and time components stored within a DataFrame is critical for successful time series analysis and aggregation tasks. A common requirement is the need to isolate specific components, such as the hour, from a complete timestamp column.

This comprehensive guide details the precise methods available within PySpark to achieve this extraction, focusing primarily on two powerful functions provided by the `pyspark.sql.functions` module: `F.hour()` and `F.date_trunc()`. Understanding the subtle differences between these approaches is vital, as one returns an integer representing the hour, while the other returns a new timestamp object truncated to the beginning of that hour.

The ability to accurately extract the hour component unlocks numerous possibilities for data transformation. For instance, analysts often need to determine peak usage times for web traffic, sales activity, or system events. By creating a dedicated hour column, we can easily group the data, calculate aggregate statistics, and visualize trends based on time of day, thereby turning raw timestamp information into actionable insights for business intelligence and operational optimization.

PySpark: Extract Hour from Timestamp

This section outlines the two primary approaches for isolating the hour component from a timestamp column within a PySpark DataFrame. The choice between them depends entirely on whether you require the hour as a discrete numeric value or as a time-truncated timestamp object.

Method 1: Extracting the Numeric Hour using F.hour()

The first and most direct method involves using the `hour()` function found within the `pyspark.sql.functions` module. This function is specifically designed to parse a timestamp column and return the integer value representing the hour of the day (ranging from 0 to 23). This is typically the preferred method when aggregating data or performing arithmetic based on the time slot.

To implement this, we utilize the `withColumn` transformation, which allows us to add a new column to our existing DataFrame. The new column is generated by applying `F.hour()` to the target timestamp column. Ensure you import the functions module, conventionally aliased as `F`, for concise code readability.

The following syntax demonstrates how to apply `F.hour()` to a column named `ts` and store the result in a new column also named `hour`:

```
from pyspark.sql import functions as F

df_new = df.withColumn('hour', F.hour(df))
```

For example, if the input timestamp column contains the value **2023-01-15 04:14:22**, executing this command will generate a new column entry containing the integer **4**. This straightforward extraction is essential for numeric aggregation tasks.

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

The second viable method involves using the `date_trunc()` function. Unlike `F.hour()`, which returns an integer, `F.date_trunc()` returns a new timestamp value that has been precisely truncated to the boundary of the specified time unit. When instructed to truncate to the `'hour'` unit, all minute, second, and fractional second components of the original timestamp are set to zero, effectively returning the timestamp at the start of that specific hour.

This technique is particularly useful in scenarios where you need to maintain the full date context while normalizing the time component. For instance, if you want to group events by the hour they occurred, but still retain the year, month, and day information, truncation is the appropriate approach.

The implementation requires passing the desired truncation unit (`'hour'`) as the first argument to the function, followed by the timestamp column reference:

```
from pyspark.sql import functions as F

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

If the input timestamp is **2023-01-15 04:14:22**, this syntax will return the resultant timestamp **2023-01-15 04:00:00**. Notice that the date component is preserved, but the time is normalized to the start of the fourth hour.

Setting Up the Demonstration Environment

To illustrate both methods effectively, we must first establish a functional PySpark environment and create a sample DataFrame. Our demonstration uses hypothetical sales data, recording transactions at specific points in time. Before any temporal extraction can occur, it is essential that the timestamp column is correctly defined using one of Spark's supported Date and Time Data Types, specifically the `TimestampType`.

The following setup script initializes the `SparkSession`, defines our raw data containing

timestamps (initially as strings) and corresponding sales figures, and then performs a crucial data type conversion. The `F.to_timestamp()` function is used to convert the string representation into a proper timestamp format, ensuring that subsequent time extraction functions operate correctly.

Data preparation is a non-negotiable step in any data pipeline. If the input data remains a simple string, PySpark's specialized time functions will fail or produce null results. We specify the exact format (`'yyyy-MM-dd HH:mm:ss'`) to ensure accurate parsing and conversion, making the `ts` column ready for our extraction tasks.

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|
+-------------------+-----+

Detailed Implementation: Using F.hour() for Numeric Extraction

This example demonstrates the practical application of Method 1, where the objective is to isolate the hour value as a discrete integer for downstream analytical processes, such as calculating average sales per hour across the day. The resulting data will be suitable for immediate aggregation or charting based on the 24-hour cycle.

By applying `F.hour()` to the `ts` column of our prepared sales DataFrame, we generate a new column named `hour`. This transformation operates on a row-by-row basis across the distributed dataset, leveraging Spark's optimized execution engine to efficiently compute the hour value for every record. It is important to remember that this function strips away all other temporal context--the date, minutes, and seconds are ignored in the output.

Observe the code block below. The syntax is concise, highlighting the power of the `pyspark.sql.functions` library in achieving complex data manipulations with minimal code. The output shows the new `hour` column populated with integers corresponding to the time of transaction.

from pyspark.sql import functions as F

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

#view new DataFrame
df_new.show()

```
+-------------------+-----+----+
| ts|sales|hour|
+-------------------+-----+----+
|2023-01-15 04:14:22| 225| 4|
|2023-02-24 10:55:01| 260| 10|
|2023-07-14 18:34:59| 413| 18|
|2023-10-30 22:20:05| 368| 22|
+-------------------+-----+----+
```

The resulting **hour** column successfully holds only the integer hour (0-23) derived from the original timestamp in the **ts** column. For instance, the first record, which occurred at 4:14:22 AM, yields the hour value 4, and the transaction at 10:55:01 AM yields 10, clearly demonstrating the function's intended behavior.

Detailed Implementation: Using F.date_trunc() for Timestamp Normalization

This second example applies Method 2, which utilizes `F.date_trunc()`. This function serves a different purpose than `F.hour()`; instead of extracting an integer, it aligns the entire timestamp to the nearest specified boundary. When truncating to the hour, the function ensures that all resulting timestamps reflect the start of the hour in which the original event occurred, effectively standardizing the data point.

Truncation is frequently employed when the goal is aggregation by time window while maintaining the integrity of the date component. This is often necessary for analysis that involves cross-referencing activity across multiple days or weeks, preventing the loss of the date information that would occur if only the integer hour were extracted.

In the following script, we apply `F.date_trunc('hour', df)` to generate the normalized timestamp. Note that although the new column is also named `hour` in this demonstration for simplicity, its data type remains `TimestampType`, distinct from the integer output of the previous method.

from pyspark.sql import functions as F

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

#view new DataFrame
df_new.show()

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

The new **hour** column shows each timestamp from the **ts** column truncated to the hour, demonstrating that the date component is preserved, but the minutes and seconds are reset to 00:00. This is the ideal output format when aggregating events into discrete hourly bins while preserving the day context.

Choosing the Optimal Extraction Method

Determining whether to use `F.hour()` or `F.date_trunc()` depends critically on the subsequent analytical steps you plan to execute. Both methods are highly efficient within PySpark, but their resulting data formats cater to different analytical needs. Making the correct choice optimizes both code clarity and computational efficiency.

If your primary goal is to study cyclical patterns that repeat daily--such as identifying the peak hour of activity regardless of the specific day--then `F.hour()` is the superior choice. It generates a small integer data type, which is computationally lightweight and perfectly suited for grouping operations (e.g., `df.groupBy('hour').agg(F.sum('sales'))`). This approach is best for pure time-of-day analysis.

Conversely, if the analysis requires maintaining the specific date associated with the time window--for instance, charting hourly sales trends over a full calendar year where each hour needs to be uniquely identified by both date and time--then `F.date_trunc()` is mandatory. Truncation provides a normalized timestamp boundary that respects the day, month, and year, enabling correct chronological ordering in large-scale time series analysis.

Performance Considerations for Large Datasets

When dealing with extremely large DataFrames, performance differences, though often minor, can become relevant. Both `F.hour()` and `F.date_trunc()` are highly optimized built-in functions that execute their logic directly on the Java Virtual Machine (JVM) using Apache Spark SQL engine, minimizing the overhead typically associated with Python operations.

However, extracting an integer (`F.hour()`) generally involves a marginally simpler computation than generating a full new timestamp object (`F.date_trunc()`) and maintaining its structure. Furthermore, the resulting DataFrame using `F.hour()` will utilize less memory because an integer column occupies less space than a timestamp column. For analyses where memory footprint and aggregation speed are paramount, and the date context is unnecessary, the integer extraction method should be favored.

It is always recommended, especially in production pipelines, to examine the execution plan (using `df.explain()`) to ensure the operations are being pushed down efficiently to the underlying Spark engine. Fortunately, for standard functions available in `pyspark.sql.functions`, such optimization is standard practice, providing reliable and scalable performance regardless of the method chosen.

Summary of Key Functions

We have successfully explored the two main approaches for dealing with temporal granularity at the hourly level in PySpark. To ensure clarity, here is a concise summary of the utility and outcome of the discussed functions:

F.hour(column): Extracts the numerical hour (0-23) from a timestamp. Output type is `IntegerType`. Best suited for daily cyclical analysis and lightweight grouping.

F.date_trunc('hour', column): Returns the full timestamp truncated to the beginning of that specific hour (minutes and seconds set to 00:00). Output type is `TimestampType`. Necessary for preserving date context during hourly grouping.

F.to_timestamp(column, format): Crucial for converting raw string data into the proper `TimestampType` required by all other time functions in PySpark. This preparatory step ensures data validity.

Conclusion and Next Steps

The extraction of the hour component from a timestamp in PySpark is a straightforward task, facilitated by the robust collection of functions available in `pyspark.sql.functions`. Whether you require the hour as a simple integer for cyclical analysis using `F.hour()` or a normalized timestamp for detailed chronological grouping using `F.date_trunc()`, Apache Spark provides the tools necessary for efficient, distributed processing.

By implementing the demonstrated techniques, analysts can transform raw transactional data into structured, time-aware datasets, enabling sophisticated statistical analysis and visualization. Always remember the critical prerequisite: ensuring your source column is correctly cast to the `TimestampType` before attempting any extraction or truncation operations.

To further enhance your skills in managing temporal data within the PySpark ecosystem, we recommend exploring related functions that handle other temporal components, such as `F.dayofweek()`, `F.month()`, and `F.unix_timestamp()`. These functions collectively provide a powerful toolkit for comprehensive time-based data manipulation.

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