Learning PySpark: Extracting the Hour from Timestamp Data

Authored by

Mohammed looti

November 11, 2025

Mastering Temporal Data Extraction in PySpark

Efficiently processing time-series data is a cornerstone of modern data engineering pipelines. Handling complex temporal components, such as the timestamp, with speed and accuracy is non-negotiable for any analytical workflow. When dealing with massive, distributed datasets, PySpark offers specialized, highly optimized functions designed to manipulate datetime objects seamlessly within a DataFrame. Isolating specific temporal attributes, such as the hour, is often the first step in critical tasks, including grouping data for hourly metrics, calculating peak usage times, or identifying subtle temporal patterns in user activity or sales trends. This comprehensive guide will detail the two primary, highly performant methods available in PySpark SQL functions that allow developers to extract or normalize the hour component from a source timestamp column.

The choice between these two methods hinges entirely upon the intended analytical goal and the required output format. Do you require the hour represented as a simple **integer** (ranging from 0 to 23) suitable for numerical comparison and machine learning features? Alternatively, do you need the original timestamp value **truncated** (normalized) to the nearest hour, thereby preserving the date context while resetting minutes, seconds, and milliseconds to zero? Both approaches are fundamentally important, but they serve distinct purposes: integer extraction is ideal for filtering and classification based on the time of day, whereas truncation is vital for robust time-series aggregation and grouping. We will meticulously explore the necessary syntax, practical usage, and core implications of both techniques, ensuring you can deploy the most appropriate solution within your complex data processing environment.

Essential Prerequisites and Setup Requirements

Before attempting any form of temporal manipulation in PySpark, a mandatory prerequisite is confirming that the column containing your temporal data is correctly interpreted as a **TimestampType**. The performance and reliability of PySpark's execution engine rely heavily on accurate schema metadata to execute its optimizations efficiently. If your source data--perhaps ingested from a CSV or JSON file--is currently stored as a string (e.g., 'YYYY-MM-DD HH:MM:SS'), you must first perform a rigorous conversion. This initial step utilizes functions like `F.to_timestamp()`, ensuring the data conforms to the expected **TimestampType**. This conversion is a critical preliminary step; skipping it will invariably lead to runtime errors, null values, or incorrect results when using specialized datetime functions.

All the powerful manipulation methods demonstrated throughout this guide depend on correctly importing the necessary functions from the SQL module. Following standard community practice, these functions are conventionally aliased as `F` (short for functions) to enhance code brevity and readability within the PySpark environment. Furthermore, the subsequent code examples assume that a PySpark DataFrame, conventionally denoted as `df`, has already been successfully initialized

and contains a valid timestamp column named `ts`. A solid understanding of how these functions integrate with the DataFrame API--specifically through the non-mutating `.withColumn()` transformation--is crucial for implementing these operations successfully and maintaining the integrity of the original dataset structure.

Method 1: Extracting the Hour as an Integer (F.hour)

The most direct and frequently used approach for retrieving the hour component as a discrete numerical value (spanning the range 0 to 23) is through the dedicated PySpark built-in function: `F.hour()`. This function is specifically engineered to isolate and return only the hour field from a properly formatted timestamp column. This technique is especially advantageous when the primary goal is to perform operations like bucketing events into 24 distinct time slots, analytically calculating peak hour utilization, or profiling the overall distribution of activities throughout a 24-hour cycle. The resulting output is an **integer column** (IntegerType), which is highly efficient for subsequent numerical computations, filtering, and feature engineering tasks.

To seamlessly apply this function, we employ the standard `.withColumn()` DataFrame transformation. This transformation creates an entirely new column--which we will label `hour` in our examples--and populates every row by applying the `F.hour()` function directly to the values within the source timestamp column (`ts`). This process ensures the original column remains untouched while generating the desired feature.

The following concise code snippet demonstrates this operation. If the input timestamp value in the source column is **2023-01-15 04:14:22**, the resulting value inserted into the new `hour` column will be the integer **4**, which precisely represents the fourth hour of the day according to the 24-hour clock.

from pyspark.sql import functions as F

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

The inherent efficiency of `F.hour()` stems from its optimized, direct extraction mechanism. It completely bypasses the need for complex, often costly, string manipulation or formatting operations that are known to be significantly less performant when scaled across vast datasets. Consequently, this method is unequivocally the preferred solution whenever the minute, second, or millisecond components of the timestamp are irrelevant to the current analytical task, and only the pure numerical hour is required.

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

A common requirement, particularly in complex aggregation tasks, extends beyond simply obtaining the hour number. Often, data scientists need to normalize the original timestamp itself. This involves modifying the timestamp so that all events that occurred within the same 60-minute window are assigned an identical, standardized timestamp value. This powerful normalization technique is accomplished using the `F.date_trunc()` function. This function is designed to truncate a datetime value down to a specified unit of precision--in this specific case, `'hour'`--thereby effectively resetting all finer time components (minutes, seconds, milliseconds) to zero.

Leveraging `F.date_trunc()` is indispensable for time-series aggregation workflows, such as calculating the total count of events or the sum of sales figures recorded within every single clock hour. By standardizing the timestamp column, we successfully create a consistent, common key suitable for highly efficient grouping operations. For instance, if you are attempting to calculate the total sales volume for every hour, truncating the timestamp ensures that all sales recorded between 04:00:00 and 04:59:59 are accurately grouped under the single, normalized marker value of 04:00:00.

To illustrate the impact, if the original timestamp input is **2023-01-15 04:14:22**, applying the truncation operation precisely to the hour level results in the standardized output timestamp **2023-01-15 04:00:00**. This output is critical because it successfully maintains the full contextual date information while simultaneously providing the clean, standardized hourly marker necessary for aggregation.

from pyspark.sql import functions as F

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

It is vital to recognize the fundamental difference in the resulting column type: Method 1 (`F.hour()`) reliably produces an **IntegerType**, whereas Method 2 (`F.date_trunc()`) returns a **TimestampType**, albeit one that has been deliberately modified and normalized. This crucial distinction must be factored into the planning of any subsequent SQL queries, DataFrame transformations, or Python operations performed on the resulting feature column.

Practical Demonstration: Preparing the Data

To fully demonstrate the utility and distinction of these two extraction methods, we must first establish a representative sample PySpark DataFrame. This synthetic DataFrame will model typical sales data, incorporating a timestamp column (`ts`) and an associated numerical sales amount column (`sales`). We begin by initializing the Spark session, defining our raw data, and critically, ensuring that the `ts` column--which is initially derived from simple Python strings--is correctly and robustly converted into the proper TimestampType format before any advanced

extraction functions are applied. This careful preparation step guarantees the integrity and robustness of our subsequent demonstration.

We specifically utilize the `F.to_timestamp('ts', 'yyyy-MM-dd HH:mm:ss')` function to manage this conversion. By explicitly specifying the precise format mask, we ensure that PySpark correctly interprets the string values. This conversion step, while seemingly minor, is frequently overlooked but remains mandatory for accurate and error-free datetime processing at scale.

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

from pyspark.sql import functions as F

# Define sample data containing timestamps and sales figures
data = ,
,
,
]

Define column names
columns =

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

# Convert string column 'ts' to proper 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|
+-------------------+-----+

The resulting DataFrame, now correctly labeled `df`, is fully formatted and ready for the subsequent extraction operations. We can visually confirm the presence of varied timestamp values associated

with each sales record, ranging across different dates and times from early morning (4 AM) to late evening (10 PM).

Applying Method 1: Integer Hour Extraction in Practice

We will now apply the `F.hour()` function to our meticulously prepared DataFrame to clearly demonstrate the process of extracting the hour as a discrete integer. This output is profoundly useful for immediate analytical needs that require numerical comparisons or categorization based purely on the time of day, completely independent of the specific date. We generate a new DataFrame, `df_new`, which seamlessly incorporates the extracted hour into a dedicated column named `hour`.

Carefully observe how the function successfully and accurately maps the complete timestamp (e.g., 2023-01-15 04:14:22) to its corresponding integer hour value (4). This transformation is a powerful technique for simplifying complex datetime fields into straightforward numerical features, which are perfectly optimized for high-speed aggregation or integration into sophisticated machine learning models.

from pyspark.sql import functions as F

# Extract hour (0-23) from the 'ts' column
df_new = df.withColumn('hour', F.hour(df))

# View the new DataFrame showing the integer hour
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 newly generated `hour` column unequivocally displays the 24-hour numeric representation for the time of each sale. This confirms that `F.hour()` provides a clean, integer-based feature derived directly from the source timestamp data, satisfying all requirements for simple numerical time-of-day analysis.

Applying Method 2: Timestamp Truncation in Practice

In the final demonstration, we apply the `F.date_trunc()` function to achieve the critical goal of hourly normalization. This demonstration operates on the same source DataFrame but produces a fundamentally different and essential result: a new timestamp column where the minutes and seconds have been definitively reset to zero, effectively establishing the exact start time of the hour in which the original event took place.

When deploying this function, we explicitly define the unit of truncation as the string literal `'hour'`. This technique is absolutely foundational for all aggregation tasks because it generates a perfectly uniform key for grouping all records that fall within the same 60-minute interval. For example, if we subsequently apply a grouping operation based on this new column, every event recorded between 10:00:00 and 10:59:59 will be summarized and counted under the single, standardized 10:00:00 key.

from pyspark.sql import functions as F

# Create new column containing timestamp truncated to the hour
df_new = df.withColumn('hour_truncated', F.date_trunc('hour', df))

# View the new DataFrame showing the truncated timestamp
df_new.show()

+-------------------+-----+-------------------+
| ts|sales| hour_truncated|
+-------------------+-----+-------------------+
|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|
+-------------------+-----+-------------------+

As definitively verified by the output above, the new `hour_truncated` column successfully preserves the original date context while simultaneously resetting the time components below the hour level. This powerful and optimized technique is absolutely indispensable for accurately generating hourly aggregates and for visualizing time-series data grouped by precise, fixed intervals in PySpark.

Summary and Next Steps

The reliable extraction of the hour component from a timestamp in PySpark can be effectively

accomplished using one of two highly efficient methods: `F.hour()` for generating a pure numerical integer output, or `F.date_trunc()` for producing a standardized, normalized timestamp output. Both functions expertly leverage the optimized, distributed capabilities of the Spark SQL engine, thereby guaranteeing high performance and reliability even when processing petabyte-scale data volumes. A clear and comprehensive understanding of the functional difference between these two distinct methods is paramount, enabling developers and data scientists to select the single most appropriate tool for their specific analytical objectives, whether those objectives involve simple time-of-day filtering or complex, large-scale temporal aggregation.

For continuing professional development and deeper exploration into PySpark's extensive capabilities in handling complex data types and intricate transformations, particularly concerning temporal data, we strongly recommend consulting the official documentation and related in-depth tutorials. Mastering these core datetime functions is the fundamental key to achieving truly efficient and scalable large-scale data processing.

The following resources provide further guidance on common PySpark tasks:

Official PySpark SQL Functions Documentation
Spark SQL and DataFrames Programming Guide
Understanding Time Series Data Analysis