PySpark: Select Rows by Index in DataFrame

Authored by

stats writer

November 17, 2025

When transitioning from tools like Pandas to the scalable environment of PySpark, developers often encounter a fundamental conceptual difference: the absence of a built-in, sequential index in a standard PySpark DataFrame. This is by design, reflecting the nature of distributed computing where data partitions are processed in parallel, making a single, globally ordered sequence inherently challenging and expensive to maintain.

However, practical data manipulation tasks frequently require selecting rows based on their order or specific positional indices--for instance, pulling the first ten records or selecting a range of rows. While a direct equivalent to Pandas' `iloc` function does not exist, we can efficiently simulate this behavior by introducing an explicit index column. This guide details the expert method for generating a sequential index using window functions and subsequently leveraging standard filtering techniques to select rows based on these index values.

The Challenge of Indexing in PySpark

A critical characteristic of the PySpark DataFrame is its immutability and its distribution across a cluster of nodes. Unlike local, single-machine data structures that rely on a guaranteed row order (like Pandas or R data frames), PySpark prioritizes parallel processing. If Spark were forced to assign a strict, sequential index from 0 to N every time a DataFrame was created or modified, it would introduce a mandatory global shuffle, severely bottlenecking performance in large-scale operations.

Therefore, when we talk about selecting rows by index in PySpark, we are typically referring to selecting rows based on a **synthetic column** that we explicitly add to enforce order. This synthetic index provides the necessary positional reference without relying on the physical ordering of the data partitions managed by the underlying Apache Spark engine. Understanding this distinction is crucial for optimizing workflows and avoiding performance pitfalls in large-scale data engineering.

Prerequisites: Setting up the PySpark Environment

Before diving into the indexing mechanism, we must ensure the PySpark environment is properly initialized. All operations within PySpark require an active **SparkSession**, which serves as the entry point to communicate with the Spark cluster. We will use standard methods to import necessary modules and establish this session.

The following setup code initializes the session and prepares the environment for the subsequent creation and manipulation of our sample data. This is the foundational step for any practical application involving DataFrame operations, ensuring we can utilize powerful functions like `row_number` and `Window` for creating our index.

Step 1: Creating the Sample PySpark DataFrame

To illustrate the selection process, let us begin by defining and constructing a simple PySpark DataFrame. This DataFrame contains basic information about teams, their conferences, and a points total. The goal is to successfully assign a sequential index to these rows and then use that index for targeted filtering.

We import the `SparkSession`, define our dataset (`data`) and column names (`columns`), and then invoke `spark.createDataFrame()`. Observe the output of `df.show()`; notice that currently, there is no explicit row number associated with the data points.

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

#define data
data = ,
,
,
,
,
]

#define column names
columns =

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

#view DataFrame
df.show()

+----+----------+------+
|team|conference|points|
+----+----------+------+
A	East	11
A	East	8
A	East	10
B	West	6
B	West	6
C	East	5
+----+----------+------+

Step 2: Generating a Sequential Index Column using row_number()

To create a reliable, sequential index ranging from 1 to N (where N is the total number of rows), we must utilize a Window function combined with the row_number() analytical function. The `row_number()` function assigns a sequential integer value to each row within a partition. Since we want a global sequence (not partitioned), we define a window over the entire PySpark DataFrame.

The key technical detail here involves the `Window().orderBy()` clause. While analytical functions typically require an ordering to define row sequence, if we are only concerned with creating an arbitrary global index (i.e., we don't care which row gets '1' and which gets '2' as long as the sequence is continuous), we can use a dummy ordering. We use lit('A')--the literal function--to provide a constant value for ordering. This approach ensures that Spark can efficiently assign unique, sequential numbers across the entire dataset without requiring a complex sort based on existing column values.

We wrap the window specification in a variable `w` and then apply `row_number().over(w)` within the `withColumn` operation, naming our new index column `id`. This newly created `id` column serves as our explicit, sequential index for filtering purposes.

from pyspark.sql.functions import row_number,lit
from pyspark.sql.window import Window

```
#add column called 'id' that contains row numbers from 1 to n
w = Window().orderBy(lit('A'))
df = df.withColumn('id', row_number().over(w))

#view updated DataFrame
df.show()

+----+----------+------+---+
|team|conference|points| id|
+----+----------+------+---+
| A| East| 11| 1|
| A| East| 8| 2|
| A| East| 10| 3|
| B| West| 6| 4|
| B| West| 6| 5|
| C| East| 5| 6|
+----+----------+------+---+
```

Step 3: Selecting Rows Using Index Ranges (where and between)

Once the sequential index column (`id`) is established, we can treat it like any other column in the DataFrame and apply standard SQL-like filtering operations. A common use case for index-based selection is retrieving a continuous subset of rows, such as the second through the fifth row.

We accomplish this using the where() function (which is synonymous with `filter()`) in conjunction with the powerful `between()` function. The `between()` function allows us to specify an inclusive range, making it ideal for selecting blocks of records defined by their index boundaries. We must import the `col` function to reference the `id` column effectively within the filter expression.

The following code snippet demonstrates filtering the DataFrame to include only those rows where the `id` value falls inclusively between 2 and 5. This method is highly expressive and performs well, leveraging Spark's optimized filtering capabilities on the explicit index column.

from pyspark.sql.functions import col

#select all rows between index values 2 and 5
df.where(col('id').between(2, 5)).show()

+----+----------+------+---+
|team|conference|points| id|
+----+----------+------+---+
A	East	8	2
A	East	10	3
B	West	6	4
B	West	6	5
+----+----------+------+---+

As demonstrated in the output, the resulting DataFrame successfully isolates the subset of records corresponding to index values 2, 3, 4, and 5 from our generated index column. This technique provides the precise positional slicing capabilities often desired in data manipulation tasks.

Step 4: Selecting Specific Rows Using Index Values (filter and isin)

Beyond selecting continuous ranges, there is often a need to select non-contiguous rows based on a specific list of indices. For example, if we needed the very first, fifth, and last row of the dataset, selecting by range would be inefficient or impossible. For this scenario, the combination of the `filter()` function and the `isin()` operator is the ideal solution.

The `isin()` operator checks whether a column's value is present within a provided set of values.

By applying this operator to our `id` column, we can specify exactly which indices we wish to retrieve, regardless of their numerical proximity to one another. This provides highly granular control over the row selection process based on the positional index we previously created.

In this example, we target the rows corresponding to index positions 1, 5, and 6. This illustrates the flexibility of using the explicit index column for advanced, non-sequential filtering. Note that we can use either `filter()` or where() interchangeably for this operation.

#select specific rows based on a list of index values
df.filter(df.id.isin(1,5,6)).show()

```
+----+----------+------+---+
|team|conference|points| id|
+----+----------+------+---+
| A| East| 11| 1|
| B| West| 6| 5|
| C| East| 5| 6|
+----+----------+------+---+
```

The resulting output displays the rows in the DataFrame in index positions **1**, **5** and **6**, confirming that only the rows corresponding to the designated index positions are included in the final filtered DataFrame. This robust methodology allows PySpark users to implement complex positional selection logic essential for common data science tasks like sampling or validation splitting.

Alternative Indexing: Using monotonically_increasing_id()

While row_number() guarantees a sequential index starting from 1 (or 0, depending on implementation specifics), another commonly cited function for index generation in PySpark DataFrame is `monotonically_increasing_id()`. It is important to understand the difference between these two approaches, as they serve distinct purposes regarding indexing.

The `monotonically_increasing_id()` function generates ID numbers that are guaranteed to be non-decreasing, and they are unique within the DataFrame. Crucially, they do **not** guarantee sequentiality, nor do they guarantee starting at 0 or 1. The IDs are constructed using the partition ID and the record index within that partition. This function is extremely fast and efficient because it does not require a global sort or shuffle, aligning perfectly with the principles of distributed computing.

If the absolute sequential ordering of rows (1, 2, 3, 4, ...) is mandatory for your filtering logic (e.g., selecting the true "first 10 rows" based on some inherent but undefined order), then row_number() is necessary, accepting the performance cost of the required ordering operation. If you only require

unique IDs for subsequent joins or mapping, and strict sequentiality is not required, `monotonically_increasing_id()` is the preferred, high-performance option.

Performance Considerations and Best Practices

Creating an explicit index column in a PySpark DataFrame, especially using the row_number() approach, has a significant performance implication that users must recognize. Since `row_number()` requires a global ordering defined by the Window function (even if the ordering criterion is arbitrary, like lit('A')), Spark must collect all data onto a limited number of executors--potentially one--to ensure the sequence is strictly incremental across all partitions. This operation is known as a **global sort** and can create a severe bottleneck for very large datasets.

Therefore, the best practice is to avoid generating a strict sequential index unless it is absolutely essential for the business logic. If you only need to segment or sample the data, consider using randomized sampling techniques or hash functions instead of strict positional indexing. If positional indexing is unavoidable, ensure this operation is performed as late as possible in your ETL pipeline, preferably after significant data reduction or filtering has occurred, minimizing the volume of data subject to the costly global sort.

When filtering using the index, Spark's query optimizer efficiently uses the indexed column for selection. Both `where(col('id').between(X, Y))` and `filter(df.id.isin(A, B, C))` are highly optimized filter operations once the index is established, ensuring swift retrieval of the desired rows. Always utilize the native PySpark SQL functions (like `lit`, `col`, `between`, `isin`) rather than converting the DataFrame to Pandas or using User Defined Functions (UDFs) for filtering, as the former allows the Spark engine to perform predicate pushdown and execution optimization.

Summary of Indexing Techniques

Selecting rows by index in PySpark requires a deliberate shift from the index-centric mindset of single-node computing to a distributed paradigm. Here is a brief summary of the methods discussed:

Simulating Sequential Index: Use `row_number().over(Window().orderBy(lit('X')))` to create a sequential, 1-to-N index (`id`). This is the standard solution for strict positional requirements.

Selecting Ranges: Use `df.where(col('id').between(Start, End))` to extract a continuous block of rows.

Selecting Specific Indices: Use `df.filter(df.id.isin(I1, I2, I3))` to extract non-

contiguous rows based on a list of specific index numbers.

Alternative for Uniqueness: Use `monotonically_increasing_id()` if uniqueness across the cluster is required but strict sequential order is not, providing superior performance.

By mastering these techniques, developers can effectively manage positional data requirements within the high-performance constraints of the Apache Spark framework, transforming what appears to be a limitation into a robust, scalable feature suitable for large-scale data processing tasks.