

The Research Project

on

Apache Beam

Introduction

Apache Beam is an open-source, unified programming model designed to process large-scale, unbounded, and batch data processing pipelines. It provides a simple yet powerful API for building and executing data processing pipelines that can run on a variety of execution engines such as Apache Flink, Apache Spark, and Google Cloud Dataflow.

It works on the velocity feature of the big data. The data that is collected is dynamic data and it reduces the latency in collection of the dynamic data. The concept of windowing involved helps to reduce the lag to minimum. The windows are timed to last for say 30 minutes from the time of last response received that helps to pick data that is generated closely within the 30 minutes range from each other. The windowing can be done based on other features besides time. Since there is an interest in the Dynamic data therefore, we look into the time factor for windowing.

To install the Apache Beam Python SDK, you can follow these steps:

Install Python 3.x: Make sure you have Python 3.x installed on your machine. You can download it from the official Python website: <https://www.python.org/downloads/> .

Install pip: Pip is the package installer for Python. You can check if pip is installed by running the command `pip --version` in your terminal. If it is not installed, you can download it from <https://pip.pypa.io/en/stable/installing/> .

Install Apache Beam: Once you have pip installed, you can install Apache Beam by running the command `pip install apache-beam` in your terminal.

Run following commands in the command prompt as an administrator:

```
pip install apache-beam
```

Verify the installation: To verify that Apache Beam has been installed successfully, run the following command in your terminal: `python -c "import apache_beam as beam; print(beam.__version__)"`. This prints the version number of Apache Beam that you have installed.

```
C:\Windows\System32>python -c "import apache_beam as beam; print(beam.__version__)"
2.46.0
```

Where to use Apache Beam: It is useful for write a logic only once and run it on different platforms like spark, flink, data flow etc. IT gives the flexibility to port the same logic to multiple execution platform.

Provides a unified programming model for batch and stream processing. Supports multiple languages such as Java, Python, Go, and others. Supports multiple execution engines, allowing for portability of pipelines across different systems. Provides advanced features such as windowing, triggers, and side inputs. Offers a flexible and scalable architecture for handling large volumes of data.

It is used to reduce latency of individual results.

Where we can't use Apache Beam or its limitations: The learning curve can be steep for developers new to Apache Beam. The execution of pipelines can be slow, especially when dealing with complex transformations. It may not be the best fit for applications with simple data processing requirements.

The construction of a pipeline

Apache Beam pipelines consist of several components that work together to process data. Here are the key components of an Apache Beam pipeline:

Data Source: This is where the data originates from, such as a database, file system, or streaming service.

PTransforms: These are the data processing operations that transform the data as it flows through the pipeline. PTransforms can be chained together to form complex data processing workflows.

PCollections: These are the data sets that are processed by the pipeline. They can represent data in a variety of formats, including bounded (static) data sets or unbounded (streaming) data sets.

Note: *The data transformation can be Parallel processing also called ParDo*, and more types of transformations include group by, flatten, combine operation on the Pcollection and partition.

Data Sink: This is where the processed data is written to, such as a database or file system.

Pipeline Options: These are configuration settings that control how the pipeline is executed, such as the number of worker nodes, the size of the data bundles, and the location of the data sink.

Execution Engine: This is the system that runs the pipeline (and are thus called Runners) and manages the distribution of work across a cluster of machines. Apache Beam supports several execution engines, including Apache Flink, Apache Spark, and Google Cloud Dataflow. These are also called runners.

“Beam runners use the classes you provide to read and/or write data using multiple worker instances in parallel. As such, the code you provide for Source and FileBasedSink subclasses must meet some basic requirements:

Serializability: Your Source or FileBasedSink subclass must be serializable. The service may create multiple instances of your Source or FileBasedSink subclass to be sent to multiple remote workers to facilitate reading or writing in parallel. The way the source and sink objects are serialized is runner specific.

Immutability: Your Source or FileBasedSink subclass must be effectively immutable. You should only use mutable state in your Source or FileBasedSink subclass if you are using lazy evaluation of expensive computations that you need to implement the source.

Thread-Safety: Your code must be thread-safe. The Beam SDK for Python provides the RangeTracker class to make this easier.

Testability: It is critical to exhaustively unit-test all of your Source and FileBasedSink subclasses. A minor implementation error can lead to data corruption or data loss (such as skipping or duplicating records) that can be hard to detect. You can use test harnesses and utility methods available in the source_test_utils module to develop tests for your source.”

(Source: <https://beam.apache.org/documentation/io/developing-io-python/>)

It also deploys a windowing concept where we can define a time window on which one can run their aggregations. That is useful when collecting dynamic data. The aggregation happens on the mentioned windowing time frame. Then we have Triggers that define when we want to run our aggregates/logic. Trigger can be time based, data based or processing based. Then we have Schema that can be applied on the Pcollection.

Windowing is a key feature of Apache Beam that allows developers to divide a data stream into logical groups called windows. Windows are used to group data into fixed or sliding time intervals or other custom criteria, such as session-based windows. The windowing mechanism in Apache Beam allows for efficient and accurate processing of streaming data.

Here are the different types of windowing techniques that can be used in Apache Beam:

Fixed Windows: Data is grouped into fixed-size windows based on a specified time interval, such as 1 minute or 5 minutes.

Sliding Windows: Data is grouped into sliding windows that overlap with each other. For example, you could have a sliding window that moves every 5 seconds, and a window size of 30 seconds, so each window would include the last 30 seconds of data, with 25 seconds of overlap between each window.

Session Windows: Data is grouped into windows based on gaps between events, such as a period of inactivity. For example, you could have a session window that groups all events that occur within 5 minutes of each other.

Global Windows: Data is grouped into a single global window that contains all data.

When you apply windowing to a data stream in Apache Beam, each element in the stream is associated with a specific window, based on the windowing criteria you have defined. This

allows you to perform computations on data that is grouped into logical windows, which can improve the accuracy and efficiency of your data processing tasks.

Windowing can be used in conjunction with other features of Apache Beam, such as triggers and accumulation modes, to provide more advanced processing capabilities for streaming data. Overall, windowing is a powerful feature of Apache Beam that allows developers to build efficient and accurate streaming data processing pipelines.

Most of the time for Dynamic data sliding windows are used. Suppose you are building a data processing pipeline that receives real-time streaming data from an IoT device. The pipeline needs to perform some transformations on the incoming data and store the results in a database.

Using Apache Beam, you can define your pipeline as follows:

Define the data source as a streaming source, such as Apache Kafka. Use the Beam SDK to transform the incoming data by filtering out irrelevant data, aggregating values, and performing other transformations. Use a database connector to store the results in a database.

As mentioned in part second Apache Beam is used for Data Processing, that includes ETL which is Extract the data, then Transform the data and then Load the data after the transformation.

Importance/speciality: It does not rely on a particular type of execution engine. It is execution platform agnostic and data agnostic and then programming agnostic.

An explanation of methodology

Here's how an Apache Beam pipeline typically works:

Define the Pipeline: You start by defining the pipeline and its components, including the data source, transformations to be applied, and the data sink where the processed data will be stored.

Apply Transformations: Next, you apply transformations to the data in the pipeline. These transformations are operations that take input data, perform some operation on it, and produce output data. Apache Beam provides a rich set of built-in transformations, such as filtering, grouping, and aggregating.

Partition the Data: If your data is too large to fit into memory, Apache Beam will partition it into smaller chunks, called "bundles," that can be processed independently in parallel.

Distribute the Workload: Once the data is partitioned, Apache Beam distributes the workload across a cluster of machines. This allows for faster processing and scalability, as more machines can be added to the cluster as needed.

Execute the Pipeline: The pipeline is executed by the selected execution engine, which coordinates the processing of the data across the distributed cluster of machines. The execution engine is responsible for scheduling tasks, managing resources, and ensuring fault tolerance.

Store the Output: Finally, the processed data is stored in the specified data sink, such as a database or file system.

Overall, Apache Beam provides a powerful and flexible framework for building data processing pipelines that can handle large amounts of data in a scalable and distributed manner.

To create an Apache Beam pipeline and application, you will need to follow these general steps:

Define your input data source.

Define your transformations.

Define your output data sink.

Create a pipeline object.

Apply your transformations to the pipeline object.

Run the pipeline object.

Results

In this Big Data course's project, we are using below pipeline code to push the hardcoded the data into our environment.

First we import our apache_beam package: import apache_beam as beam:

The timestamp being used in the pipeline is a "Unix" timestamp and the project is being run on a "Windows system". The pipeline is a batch pipeline and is similar to one that is applied when reading from a streaming source.

The step-by-step explanation of the project's code:

```
class AddTimestampDoFn(beam.DoFn):
```

```
    def process(self, element):
```

```
        unix_timestamp = element["timestamp"] #[1]
```

```
        element = (element["userId"], element["click"]) #[2]
```

```
        yield TimestampedValue(element, unix_timestamp) #[3]
```

In the #[1] The AddTimeStamDoFn() is applied to the metadata of each element so that window function can access and use them in later point of time. This can be applied parallel to each element (for ParDo). Then each element is then transformed into a tuple while assigning a timestamp to metadata as in #[2]. This defining them in tuple gives us the leverage to use each tuple with a separate userid and in step #[3] we are assigning the first element of each tuple as key for that tuple.

```
timestamped_events = events | "AddTimestamp" >> beam.ParDo(AddTimestampDoFn())
```

beam.ParDo() function runs the function on each element of the tuple using the key.

```
windowed_events = timestamped_events | beam.WindowInto( Sessions(gap_size=30 * 60))
```

The window is now created with sessions of thirty (30) minutes gap size. The | operator is used to apply transformations to the pipeline.

```
sum_clicks = windowed_events | beam.CombinePerKey(sum)
```

and then the events are grouped based on their userid so as to find the sum of the clicks (this is useful when is calculating the traffic by users on to a website)

```
sum_clicks | WriteToText(file_path_prefix="output")
```

This command produces a text file called output-000000-of-000001.

The output looks like the following: (later demonstrated using screenshots)

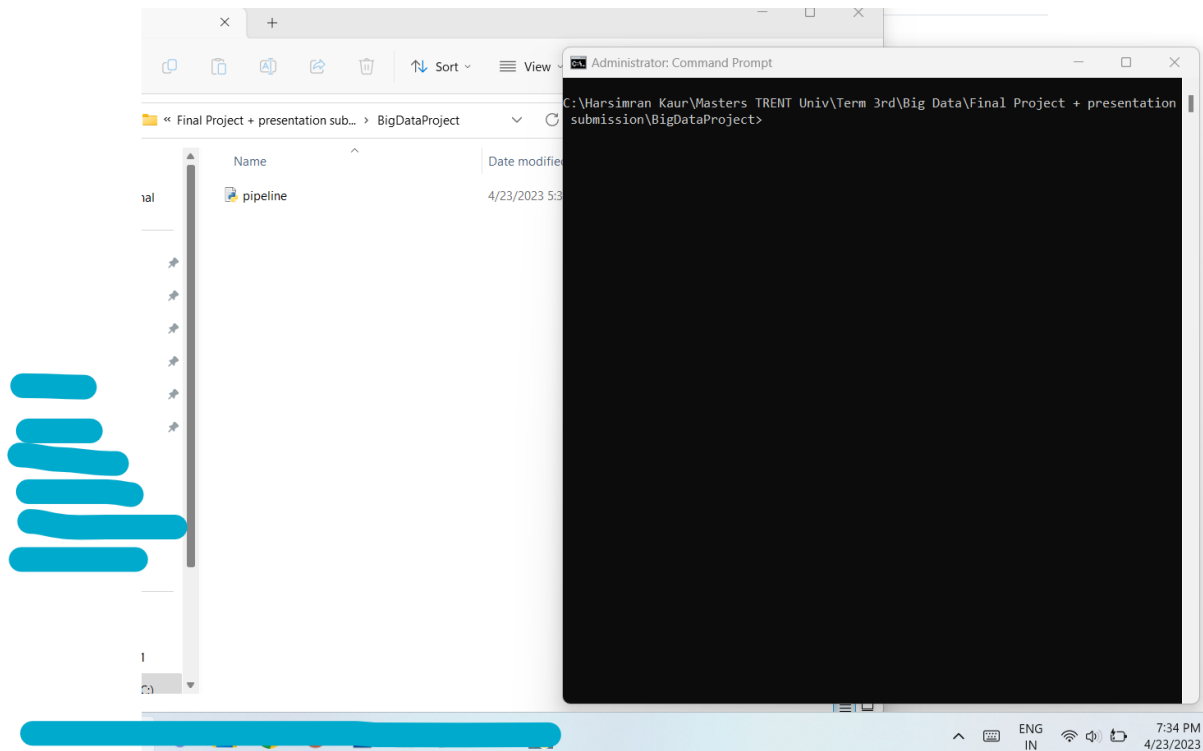
('Andy', 2)

('Andy', 1)

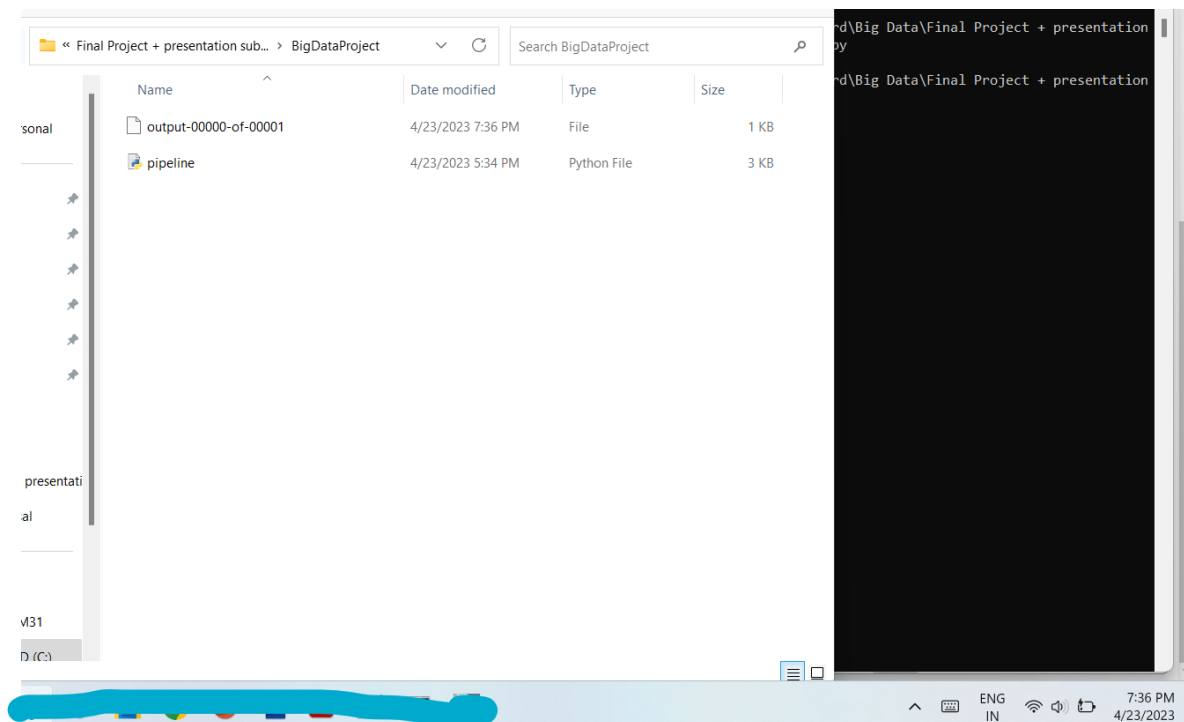
('Sam', 1)

Harsimran Kaur

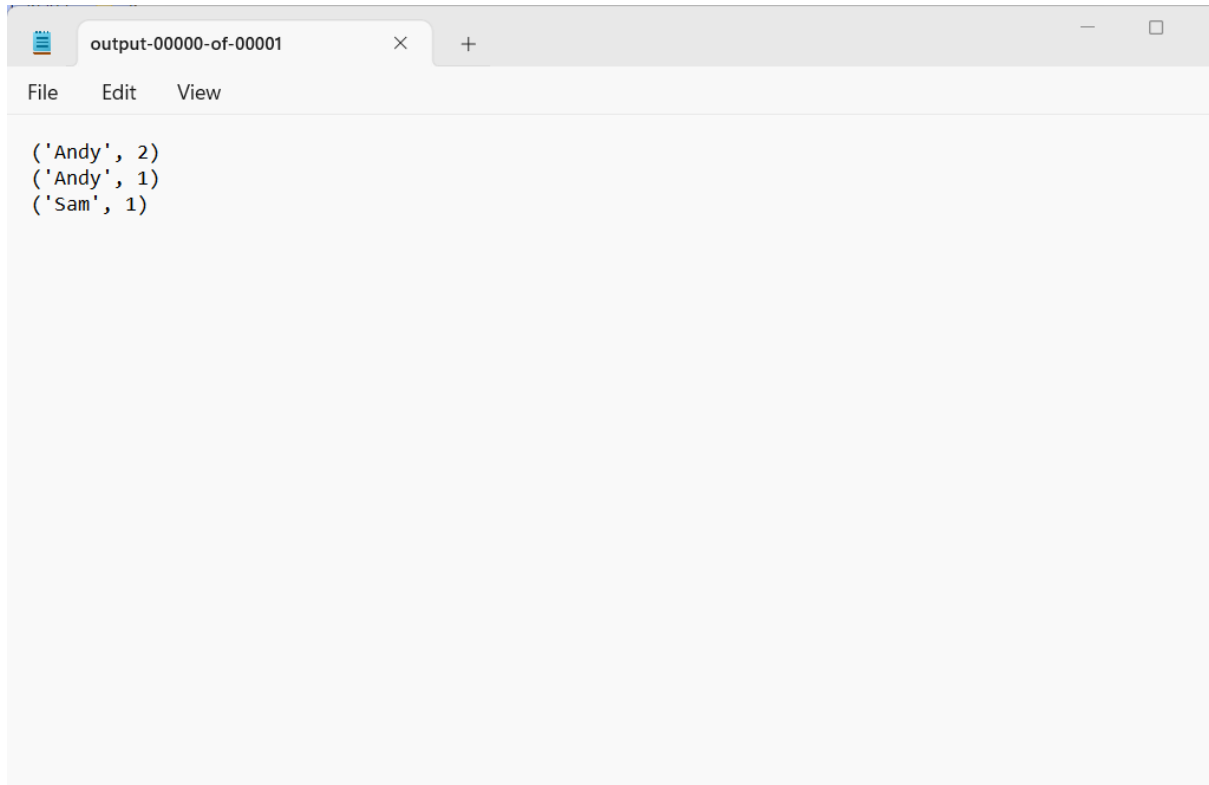
Directory structure before we run command:



Directory structure after we run the command:



Content of the output file:

A screenshot of a text editor window. The title bar shows a tab labeled 'output-00000-of-00001' with a close button (X) and a plus sign (+). The menu bar includes 'File', 'Edit', and 'View'. The text area contains three lines of code:

```
('Andy', 2)
('Andy', 1)
('Sam', 1)
```

Conclusion

The project highlights how we can use the parallel computing framework of Apache Beam to reduce latency. This can be applied when reading from a streaming source and also keeping a check on user sessions. For example, when tracking usage or access to a resource or website from multiple users or keeping a record of score of multiple users playing an online game simultaneously in a session.

So, one can setup a sliding window collection process in which the session expires or is marked complete when last response has a gap of 20 minutes (window size of 20 minutes). So, when player A clicks at 2 pm and then a player B clicks at 2:05 pm and then player clicks again at 3:00 pm while the player B clicks again at 2:15 pm, 2:30 pm and then at 2:55 pm so in that case the clicks for a sliding window for user B will 4 while for player A will be 1 until the session expires due to dormancy of more or equal to 20 minutes. This helps create entirety of the data collected for different users.

Overall, these components work together to provide a flexible and powerful platform for processing data at scale. By leveraging Apache Beam's unified programming model, developers can write data processing workflows that are portable across different execution

engines, making it easier to move their pipelines between different cloud providers or on-premises environments.

References:

<https://www.youtube.com/watch?v=waAwmY8xrgQ>

<https://www.youtube.com/watch?v=owTuuVt6Oro>

<https://yesdeepakverma.medium.com/python-apache-beam-relational-database-mysql-postgresql-struggle-and-the-solution-73e6bf3ccc28>

<https://pypi.org/project/beam-nuggets/>

<https://pypi.org/project/beam-mysql-connector/>

<https://beam.apache.org/releases/javadoc/2.46.0/index.html?org/apache/beam/sdk/extensions/sql/SqlTransform.html>

Appendix:

The code used:

```
import apache_beam as beam

from apache_beam.transforms.window import (
    TimestampedValue,
    Sessions,
    Duration,
)

from apache_beam.io.textio import WriteToText

class AddTimestampDoFn(beam.DoFn):
    def process(self, element):
        unix_timestamp = element["timestamp"]
        element = (element["userId"], element["click"])

        yield TimestampedValue(element, unix_timestamp)
```

```
with beam.Pipeline() as p:
events = p | beam.Create(
    [
        {"userId": "Andy", "click": 1, "timestamp": 1603112520}, # Event time: 13:02
        {"userId": "Sam", "click": 1, "timestamp": 1603113240}, # Event time: 13:14
        {"userId": "Andy", "click": 1, "timestamp": 1603115820}, # Event time: 13:57
        {"userId": "Andy", "click": 1, "timestamp": 1603113600}, # Event time: 13:20
    ]
)

# Assign timestamp to metadata of elements such that Beam's window functions can
# access and use them to group events.
timestamped_events = events | "AddTimestamp" >> beam.ParDo(AddTimestampDoFn())

windowed_events = timestamped_events | beam.WindowInto(
    # Each session must be separated by a time gap of at least 30 minutes (1800 sec)
    Sessions(gap_size=30 * 60),
    # Triggers determine when to emit the aggregated results of each window. Default
    # trigger outputs the aggregated result when it estimates all data has arrived,
    # and discards all subsequent data for that window.
    trigger=None,
    accumulation_mode=None,
    timestamp_combiner=None,
    allowed_lateness=Duration(seconds=1 * 24 * 60 * 60), # 1 day
)

# for each key in a collection.
sum_clicks = windowed_events | beam.CombinePerKey(sum)

# WriteToText writes a simple text file with the results.
sum_clicks | WriteToText(file_path_prefix="output")
```

The data coded into the python file:

```
{"userId": "Andy", "click": 1, "timestamp": 1603112520}, # Event time: 13:02
```

```
{"userId": "Sam", "click": 1, "timestamp": 1603113240}, # Event time: 13:14
```

```
{"userId": "Andy", "click": 1, "timestamp": 1603115820}, # Event time: 13:57
```

```
{"userId": "Andy", "click": 1, "timestamp": 1603113600}, # Event time: 13:20
```