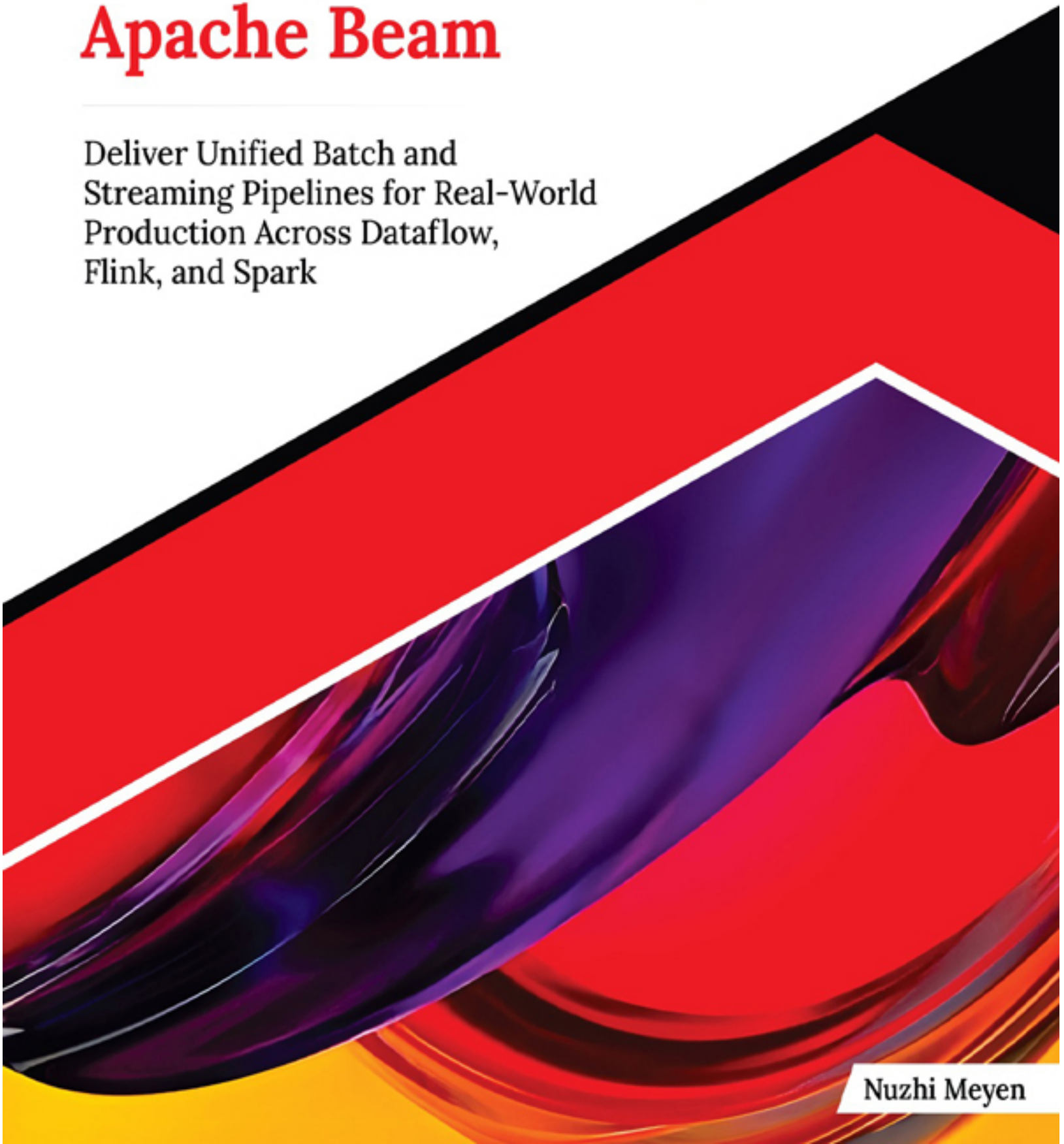
Building Data Pipelines Using Apache Beam

Deliver Unified Batch and Streaming Pipelines for Real-World Production Across Dataflow, Flink, and Spark



Nuzhi Meyen

Copyright © 2026 Orange Education Pvt Ltd, AVA®

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author nor **Orange Education Pvt Ltd** or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Orange Education Pvt Ltd has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capital. However, **Orange Education Pvt Ltd** cannot guarantee the accuracy of this information. The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

First Published: April 2026

Published by: Orange Education Pvt Ltd, AVA®

Address: 9, Daryaganj, Delhi, 110002, India

275 New North Road Islington Suite 1314 London,
N1 7AA, United Kingdom

ISBN (PBK): 978-93-49887-87-9

ISBN (E-BOOK): 978-93-49887-52-7

Scan the QR code to explore our entire catalogue

www.orangeava.com

Table of Contents

CHAPTER 1

Introduction to Apache Beam and Data Processing

Introduction

In today's data-driven world, organizations are no longer choosing between batch or streaming data—they need both, often within the same solution. From processing historical transaction logs to reacting to live events in real time, modern data platforms constantly demand flexibility, scalability, and consistency. Hence, Apache Beam is a powerful response to this need, as it offers a unified programming model that simplifies how data pipelines are designed, executed, and evolved.

Originally developed at Google and later contributed to the Apache Software Foundation, Apache Beam represents a significant shift in how data processing frameworks are approached. Rather than forcing developers to tightly couple their logic to a specific execution engine, Beam introduces a clean abstraction layer that separates *what* the pipeline does from *how* and *where* it runs. Hence, this abstraction allows the same pipeline to execute seamlessly on multiple distributed processing backends, which are referred to as **runners**, such as `Apache Spark`, `Apache Flink`, and `Google Cloud Dataflow`.

Before Apache Beam, developers had to make early, and often irreversible, architectural decisions. Batch processing frameworks like Hadoop MapReduce were well-suited for large, static datasets, but they struggled with low-latency requirements. On the other hand, stream-processing systems like Apache Storm excelled at real-time data, but they required entirely different APIs and mental models. Hence, switching between these paradigms typically meant rewriting large portions of code, retraining teams, and rethinking operational strategies.

Apache Beam addresses these challenges by offering a **single, unified API,** which is capable of handling both bounded (batch) and unbounded

(streaming) data. To better illustrate this evolution, the following table contrasts traditional approaches with the Apache Beam model.

Aspect	Traditional Batch Frameworks (for example, Hadoop)	Traditional Stream Frameworks (for example, Storm)	Apache Beam
Processing Model	Batch only	Streaming only	Unified batch and streaming
API Consistency	Batch-specific APIs	Stream-specific APIs	Single, consistent API
Portability	Tightly coupled to engine	Tightly coupled to engine	Portable across multiple runners
Switching Workloads	Requires major rewrites	Requires major rewrites	Requires no rewrites
Developer Focus	Infrastructure-heavy	Runtime-heavy	Business logic–focused

Table 1.1: *Comparing Traditional Approaches with Apache Beam*

Apache's unified API is the core reason that Apache Beam has become a foundational technology in modern data architectures. Hence, by abstracting time, windows, triggers, and execution semantics, Beam allows developers to write pipelines once. Developers can also adapt them to new platforms, scaling requirements, or deployment environments without changing the core logic.

What You Will Learn in This Chapter

By the end of this chapter, you will be able to:

- Understand the fundamental concepts behind Apache Beam's unified programming model.
- Learn how Beam pipelines differ from traditional batch and streaming approaches.
- Set up a basic development environment for Apache Beam.
- Build and run a simple **Apache Beam pipeline**, and gain hands-on exposure from the very beginning.

Reasons to Choose Apache Beam

After examining the limitations of earlier technologies and the operational friction they introduced, Apache Beam stands out as a future-proof choice.

It reduces vendor lock-in, supports evolving workloads, and aligns naturally with modern cloud and hybrid architectures. Most importantly, it allows developers and data engineers to focus on *solving business problems*, rather than rewriting pipelines every time infrastructure or processing requirements change.

For these reasons, this book focuses on Apache Beam as a foundational skill for anyone who wishes to build scalable, flexible, and long-lasting data processing systems. As you progress through the chapters, you will move through simple examples to real-world pipeline patterns. This will help you become more confident about your solutions and their ability to remain relevant, portable, and robust in a dynamic data ecosystem.

Structure

- Key Features of Apache Beam
- Understanding Beam Pipelines
- Unified Batch and Streaming Processing
- `PCollection` and `PTransform : Building Blocks of Beam`
- Building Your First Apache Beam Pipeline

Key Features of Apache Beam

Apache Beam is an open-source, unified programming model designed to simplify the development of large-scale data processing pipelines. It enables teams to build data workflows that work seamlessly across both batch and real-time streaming use cases, while it remains independent of the underlying execution engine. Therefore, by abstracting complex distributed-processing concepts behind a clean, consistent API, Apache Beam allows developers to focus on business logic rather than infrastructure details.

Conceptually, Beam guides users from defining core data processing concepts, to constructing pipelines using transforms, and finally to executing those pipelines on scalable distributed runners. Its cross-language support, portability across multiple runners, and rich ecosystem of connectors make it a powerful foundation for building scalable, future-proof data pipelines that can evolve with changing platforms and workloads. Here are the key features of Apache Beam:

- **Unified Batch and Stream Processing:** With a single API, developers can build pipelines that handle both batch and streaming data consistently.
- **Portability across Runners:** Beam allows you to write pipelines that can run on different execution engines or "runners", without changing the core business logic of the pipeline.
- **Language Support:** Apache Beam offers SDKs for multiple programming languages, including **Python**, **Java**, and **Go**, while providing flexibility for different developer needs.
- **Extensibility and Flexibility:** Through its extensive set of connectors, Beam allows easy integration with numerous data sources and sinks like Google BigQuery, Amazon S3, and Apache Kafka.

Thus, by focusing on simplicity, flexibility, and scalability, Apache Beam empowers developers to build robust data pipelines that can be deployed and run in various environments with minimal adjustments.

For **beginners**, the most important features are the **unified batch and stream processing** and **language support**, which make it easy to get started with a single, intuitive API, while **production users** benefit most from **runner portability** and **extensibility**, which enable scalable, resilient pipelines that can integrate seamlessly with enterprise-grade data platforms.

Apache Beam's Role in the Data Processing Landscape

To understand the significance of Apache Beam, it is important to recognize the current landscape of data processing. Today, data is either bounded or unbounded. Bounded data represents a finite data set, such as a collection of records in a database or log files over a certain period. Unbounded data, on the other hand, refers to an infinite stream of continuously generated data, such as real-time sensor feeds, transaction logs, or social media updates.

In the past, separate frameworks were used for batch and stream processing, each with its own strengths and limitations. For instance, batch processing tools were optimized for working on historical data, where the full data set is available from the start. On the other hand, stream processing tools are

designed for real-time data where results need to be computed as new data arrives, and thus requires more immediate feedback.

Apache Beam offers the best of both worlds. Its key value proposition lies in its ability to handle these different data types using a single, coherent programming model. Hence, whether you are processing a batch of logs from the previous day, or are working with a real-time stream of customer interactions, Beam allows you to use the same pipeline definition for both scenarios. This unified approach dramatically simplifies development efforts, and also reduces the cognitive overhead associated with maintaining multiple codebases for different processing needs. The following table summarizes how Apache Beam compares with other batch and stream processing frameworks across different dimensions such as data type to use cases, and so on.

Dimension	Batch Processing Frameworks	Stream Processing Frameworks	Apache Beam (Unified Model)
Data Type	Bounded (Finite Datasets)	Unbounded (Continuous Streams)	Bounded and unbounded
Typical Use Cases	historical analysis, ETL, and reporting	Real-time monitoring, alerts, and event processing	Historical and real-time analytics in a single pipeline
API Model	Batch-specific APIs	Stream-specific APIs	Single, unified API
Pipeline Reuse	Separate codebases required	Separate codebases required	Same pipeline definition reused
Development Complexity	Lower for batch-only needs	Higher due to time/event semantics	Simplified through unified abstractions
Operational Overhead	Multiple systems to manage	Multiple systems to manage	Reduced via consolidation
Long-Term Flexibility	Limited when requirements evolve	Limited when requirements evolve	Highly adaptable across workloads and runners

Table 1.2: *Apache Bean vs Other Batch and Stream Processing Frameworks Evaluated across Different Dimensions*

For instance, a retail company might use Apache Beam to analyze historical sales data (batch) while simultaneously processing real-time data from its online store to monitor customer activity (streaming). Beyond technical elegance, this unified approach directly enables business value, as teams

can detect purchasing trends faster, personalize customer experiences in near real time, optimize inventory levels, and respond immediately to anomalies, such as cart abandonment or fraud. Therefore, by eliminating the need for separate batch and streaming systems, Beam reduces development effort, shortens the time-to-insight, and improves operational agility. The features mentioned allow business stakeholders to act on insights proactively, rather than after the fact.

Industry evidence supports this value-driven positioning. Google Cloud customer case studies using Apache Beam–based pipelines (via Cloud Dataflow) report **30 to 50% reductions in pipeline maintenance effort**, **significant decreases in data latency**, and **faster time-to-market for analytics use cases**, particularly in retail, media, and financial services. A Forrester Total Economic Impact™ study on real-time data platforms highlights that organizations adopting unified batch-and-stream processing architectures achieve **up to 5 to 10x faster insight generation,** as well as measurable improvements in revenue uplift and cost efficiency through real-time decision-making. These outcomes demonstrate how Apache Beam is not just a technical abstraction, but a practical enabler for unlocking tangible business value from data at scale.

Use Cases for Apache Beam

The power and flexibility of Apache Beam make it suitable for a wide range of use cases across different industries. Therefore, the following examples serve as a high-level preview and will be revisited in later chapters as **hands-on case studies**, where each scenario will be explored in detail with concrete Apache Beam pipeline designs, execution strategies, and real-world considerations.

- **Real-Time Data Analytics:** Beam is often used for real-time analytics applications. For instance, financial services firms can leverage Beam to track stock market fluctuations in real time, responding to changes in seconds. Hence, by running the pipeline on a streaming runner, Beam allows analysts to process millions of market events as they happen, and it makes rapid decision-making possible.
- **Data ETL (Extract, Transform, Load):** Data pipelines often need to extract information from various sources, apply complex

transformations, and load it into a destination system. Beam simplifies the ETL process by providing built-in transforms for common tasks like filtering, mapping, and grouping data, while also being compatible with various storage systems such as **Google Cloud Storage**, **Amazon S3**, **BigQuery**, and **Apache HBase**.

For example, an e-commerce platform might need to extract customer purchase data, transform it into a structured format, and load it into a warehouse for future analysis. Beam allows them to implement this as a single pipeline, which is capable of scaling to accommodate huge volumes of data.

- **IoT Data Processing:** In industries like manufacturing and healthcare, the rise of IoT (Internet of Things) has led to the need for tools that can handle vast amounts of streaming data from devices and sensors. Apache Beam's unified approach is ideal for this, allowing companies to monitor and analyze streams of data from IoT devices while incorporating batch processing of historical data for more in-depth analysis.

  For example, in a smart factory, Beam can collect data from machinery sensors in real time, while detecting anomalies like overheating or failure based on predefined thresholds. Simultaneously, historical sensor data could be processed to identify patterns and improve long-term machine maintenance strategies.
- **Hybrid Data Pipelines:** Many organizations deal with both historical data (batch) and real-time data (streaming). Beam excels in environments where hybrid processing is necessary. For instance, social media platforms can process live user data streams to deliver real-time recommendations while running batch jobs to update user models based on historical interactions.

Hence, Beam's hybrid capabilities make it ideal for applications where consistency between batch and real-time results is critical. By eliminating the need to maintain separate codebases for batch and stream processing, Beam provides a consistent, maintainable solution for data engineers and developers alike.

The Portable Nature of Apache Beam

One of the most powerful aspects of Apache Beam is its **portability**, which means that the same pipeline code can run on multiple execution engines (runners) with minimal modifications. Runners are the distributed processing engines responsible for executing a pipeline on clusters of machines. Beam supports a wide range of runners, including:

- **`Google Cloud Dataflow`**: This is a fully managed service optimized for running Beam pipelines in the cloud.
- **`Apache Flink`**: It is a stream-first distributed processing engine, which is commonly used for real-time event processing.
- **`Apache Spark`**: It is another popular distributed processing engine, and is often used for large-scale batch and stream processing.
- **`DirectRunner`**: This is a local runner used for development and testing purposes.

This portability allows teams to build and test pipelines locally using the **`DirectRunner`** and later deploy them on a cloud platform like **`Google Cloud Dataflow`** for full-scale production processing. Apache Beam's flexibility ensures that companies can optimize their resource usage by choosing the best runner for their specific use case, whether for cost, performance, or infrastructure reasons. While Apache Beam provides strong portability, it is also important to note that not all runners support every Beam feature in exactly the same way, and portability should be viewed as *logical portability* rather than a strict lift-and-shift guarantee; the specific capabilities, limitations, and optimization considerations of each runner will be explored in detail in later chapters.

For example, a company might use **`Apache Flink`** for real-time streaming workloads, but they might opt for **`Google Cloud Dataflow`** for large-scale batch processing, all while maintaining a single codebase.

Understanding Beam Pipelines

In Apache Beam, a **pipeline** is the foundational construct used to define the series of operations that process data from input to output. Whether you are handling batch data (for example, historical records) or streaming data (for example, real-time sensor readings), all operations in Apache Beam revolve

around pipelines. A pipeline encapsulates the entire end-to-end flow, from data ingestion to transformation and storage.

This section introduces Beam pipelines and explains how they function, while covering their key components such as **`PCollection`** and **`PTransform`**, as well as providing practical examples.


Data Source
PCollection
PTransform 1
PTransform 2
PTransform 3
PCollection
Output Sink

Figure 1.1: *How a Typical Beam Pipeline Looks Like*

Defining a Beam Pipeline

At its core, a Beam pipeline represents a directed acyclic graph (DAG) of data transformations. These transformations, defined using Apache Beam's SDK, are applied to collections of data, which allows users to process both bounded and unbounded datasets in the same pipeline. In a pipeline, the user defines what happens at each stage of the process: data is ingested, transformed, and is ultimately written to an output.

Pipelines are designed to be portable across different execution environments, which implies that you can build a pipeline locally, and run it on distributed clusters with minimal changes to the codebase.

Pipeline Structure

Every Apache Beam pipeline consists of the following stages:

1. **Data Ingestion:** The pipeline starts by reading data from a source. This can be a file, a database, or a real-time data stream.
2. **Transformation:** The data is then transformed using a series of operations (such as filtering, mapping, or aggregation). Each transformation is applied to a `PCollection`, which represents the distributed data set in Beam.
3. **Output:** Finally, the pipeline writes the transformed data to an output sink, such as a file or database, or returns it for further analysis.

Simple Pipeline Example

To illustrate how a basic pipeline works, let us walk through an example. The following pipeline reads data from a text file, splits each line into words, and counts the occurrences of each word. The result is then written back to a text file.

To run this example locally, you need to set up a basic Python environment and install Apache Beam. Here are the steps to do so:

1. **Prerequisites**

    a. Python 3.8 or later

b. pip (Python Package Manager)

c. A virtual environment is recommended.

Create and activate a virtual environment through the following command:

```
python -m venv beam_env
source beam_env/bin/activate        # On Linux / macOS
beam_env\Scripts\activate           # On Windows
```

2. **Install Apache Beam**

   Install Apache Beam with the **`DirectRunner`** (default local runner):

   ```
   pip install apache-beam
   ```

3. **Prepare the Input File**

   Create a file called **`input.txt`** in the same directory as your script through:

   ```
   hello beam
   hello apache beam
   apache beam is powerful
   ```

4. **Save the Pipeline Code**

   Save your pipeline as **`wordcount.py`** through:

   ```
   import apache_beam as beam

   # Define a function to split lines into words
   def split_lines(line):
    return line.split()

   # Initialize the pipeline
   pipeline = beam.Pipeline()

   # Define the pipeline steps
   word_counts = (pipeline
         | 'ReadFromText' >>
         beam.io.ReadFromText('input.txt')
         | 'SplitIntoWords' >> beam.FlatMap(split_lines)
         | 'PairWithOne' >> beam.Map(lambda word: (word, 1))
         | 'CountPerWord' >> beam.CombinePerKey(sum)
         | 'WriteToText' >>
         beam.io.WriteToText('output.txt'))
   ```

```
# Run the pipeline
pipeline.run()
```

5. **Run the Pipeline**

   Execute this from the command line:

   ```
   python wordcount.py
   ```

   Apache Beam will use the **`DirectRunner`**, which will run everything locally on your machine.

6. **Check the Output**

   Beam will generate files such as:

   ```
   output.txt-00000-of-00001
   ```

   Inside, you will see results like:

   a. Each line is split into words using the **`FlatMap`** transform. **`FlatMap`** takes each element of a **`PCollection`** and produces zero or more elements. In this case, each line produces multiple words.

   b. The words are paired with the number 1 using the **`Map`** transform.

   c. The **`CombinePerKey`** transform then sums up all the values for each word (that is, counts the number of times each word appears).

   d. Finally, the result is written to an output text file using **`WriteToText`**.

This simple example illustrates the core concepts of Beam pipelines. Let us now break down the key components in more detail.

In Apache Beam, a **pipeline** is the central construct used to define an end-to-end data processing workflow—from reading data, to transforming it, and writing outputs. You use the pipeline to describe *what* should happen, while the selected **runner** determines *how and where* it executes (locally, or on a distributed system).

Core Abstractions: `Pipeline`, `PCollection`, and `PTransform`

Every Beam pipeline is built using three key abstractions:

- **`Pipeline`**: A **`pipeline`** is the container for your full data processing job. It acts as the entry point where you define all steps in your

workflow.

```
import apache_beam as beam
pipeline = beam.Pipeline()
```

Beam pipelines are portable, which means that the same logical pipeline can be executed in different environments by changing configuration and runner settings (for example, **`DirectRunner`** locally, **`DataflowRunner`** on Google Cloud Dataflow).

- **`PCollection`**: A **`PCollection`** represents a **distributed dataset** in Beam. All data in Beam flows through PCollections, whether the data is:
  - **Bounded**: This refers to finite data (for example, files, database extracts), which is typically found in batch pipelines.
  - **Unbounded**: This refers to infinite or continuously arriving data (for example, Kafka topics, sensor streams), which is typically found in streaming pipelines.

A **`PCollection`** is conceptually similar to a list/array, but is distributed and processed in parallel via the following command:

```
p = beam.Pipeline()
data = p | 'Create' >> beam.Create([1, 2, 3, 4, 5])
```

- **`PTransform:`** A **`PTransform`** is an operation applied to a **`PCollection`**. It can be simple (Map/Filter) or more complex (grouping/aggregation).

Two commonly used **`PTransform`** categories are:

- **Element-Wise Transforms**: These are applied independently to each element (**`Map`**, **`FlatMap`**, and **`Filter`**).
- **Key-Based Transforms**: These operate on key-value pairs (**`GroupByKey`**, **`CombinePerKey`**).

Example:

```
numbers = p | 'Create numbers' >> beam.Create([1, 2, 3, 4, 5])
squares = numbers | 'Square numbers' >> beam.Map(lambda x: x *
x)
```

Connecting Beam Concepts to Familiar Tools:

The following parallels help internalize Beam's model:

- **`Pipeline`:** Airflow DAG / Spark job / SQL query plan (defines intent; execution is handled by an engine)
- **`PCollection`**: Spark DataFrame/RDD / SQL table result / Kafka stream (distributed dataset, bounded or unbounded)
- **`PTransform`**: Spark transformations / SQL operations (**`SELECT`**, **`GROUP BY`**) / stream operators

Pipeline Execution and Runners: Once the pipeline is defined, it is executed by a **runner**, which translates the logical DAG of **`PTransforms`** into an execution plan, and schedules work on a target environment.

Common runners include:

- **`DirectRunner`**: Local execution for development/testing
- **`Google Cloud Dataflow`**: Fully managed runner for production-grade pipelines
- **`Apache Flink`**: Optimized for streaming and event-time processing
- **`Apache Spark`**: Supports both batch and streaming at scale

To execute, use the following command:

```
pipeline.run()
```

Note: *Beam pipelines are portable, but* ***not all runners support every Beam feature equally****. Thus, portability is best understood as logical portability, and runner capability differences are covered later in the book.*

Building Blocks of a Beam Pipeline

A practical pipeline typically follows three stages: **ingestion → transformation → output**.

1. **Data Ingestion**

   Read from files through:

   ```
   p = beam.Pipeline()
   lines = p | 'ReadInputFile' >>
   beam.io.ReadFromText('input.txt')
   ```

Read from cloud storage through:

```
lines = p | 'ReadFromCloud' >>
beam.io.ReadFromText('gs://bucket-name/input.txt')
```

2. **Data Transformation**

   Apply transforms to produce meaningful outputs through:

```
result = (lines
     | 'SplitIntoWords' >> beam.FlatMap(lambda line:
     line.split())
     | 'PairWithOne' >> beam.Map(lambda word: (word, 1))
     | 'CountPerWord' >> beam.CombinePerKey(sum))
```

3. **Data Output**

   Write to a text sink via the following command:

```
result | 'WriteOutput' >>
beam.io.WriteToText('output.txt')
Write to a database/warehouse (example: BigQuery):
result | 'WriteToBigQuery' >>
beam.io.WriteToBigQuery('project:dataset.table')
```

Unified Batch and Streaming Processing

Apache Beam's most powerful feature is its **unified model** for both batch and stream processing. Traditionally, developers needed to use separate frameworks or APIs for handling these types of data. **Batch processing** is optimized for finite datasets, where the full data set is known and available at the beginning of processing. **Streaming processing**, on the other hand, deals with unbounded data streams, processing data in real-time as it arrives. Apache Beam eliminates the need for separate tools, and offers a single, consistent API to handle both scenarios, whether you are processing historical data or real-time data streams. Apache Beam's most powerful feature is its unified model for both batch and stream processing. Traditionally, developers needed to use separate frameworks or APIs for handling these two types of data. Batch processing is optimized for finite datasets, where the full data set is known and available at the start of processing. Streaming processing, on the other hand, deals with unbounded data streams, and processes data in real time as data arrives. Apache Beam eliminates the need for separate tools by offering a single, consistent API to

handle both scenarios. Most importantly, the **same business logic is written once and reused without duplication**, which means that developers do not need to maintain separate codebases or introduce additional boilerplate when switching between batch and streaming execution. The pipeline definition remains unchanged; only the execution mode and data source differ, which allows teams to move seamlessly between historical and real-time processing while preserving correctness, maintainability, and development efficiency.

Hence, in this section, we will explore Apache Beam's unified model in detail, while highlighting the key differences between batch and streaming, and demonstrating how Apache Beam handles both through the same API.

Batch Processing: Finite Datasets

Batch processing is the traditional approach to data processing, and is often used for tasks like generating reports, analyzing historical data, or running large-scale ETL jobs. In batch processing, the input data is **bounded**. This means that it is a finite, predefined dataset. The system waits for all data to be available before processing begins, and the goal is to produce a complete result from the entire dataset.

Examples of Batch Processing

- **Daily Sales Reports**: A retailer might process sales data at the end of each day to generate a report summarizing total sales, broken down by store and product.
- **Data Warehousing:** Organizations regularly ingest and process batches of data from transactional databases into their data warehouses to run analytics queries.
- **ETL Pipelines:** Extract, Transform and Load (ETL) processes often run in batch mode to move and transform large datasets from one system to another.

Apache Beam supports batch processing by treating the data as a bounded `PCollection` (a fixed-size collection of elements). Once all the data is available, Beam processes it in stages defined by a series of transforms.

Example of a Batch Processing Pipeline

Let us build a simple batch pipeline that processes a set of log files and generates a report of error counts.

```
import apache_beam as beam

# Pipeline to count error messages from log files
def is_error_log(line):
 return ‹ERROR› in line

p = beam.Pipeline()

# Read from a batch data source (for example, a set of log
files)
lines = p | 'ReadLogs' >> beam.io.ReadFromText('logs/*.txt')

# Filter lines to keep only error messages
error_logs = lines | 'FilterErrors' >>
beam.Filter(is_error_log)

# Count the number of error messages
error_count = error_logs | 'CountErrors' >>
beam.combiners.Count.Globally()

# Write the count to an output file
error_count | 'WriteOutput' >>
beam.io.WriteToText('error_counts.txt')

p.run()
```

In this pipeline:

- **Bounded Data:** We are reading from a static set of log files (a bounded `PCollection`).
- **Transformations:** We filter for lines containing the word `"ERROR"`, and then count them.
- **Result:** The total number of error logs is written to an output file, once all the input data has been processed.

Batch processing is a well-suited solution for static, historical datasets that require complete and final results. However, not all data can be treated as bounded. This is where streaming processing comes in.

Stream Processing: Unbounded Data

Stream processing is essential for **handling unbounded datasets** that grow continuously over time. Unbounded data streams, such as website clickstream data, IoT sensor readings, or social media feeds, are infinite by nature. The system does not wait for the data to be "complete" because new data is always arriving.

Unlike batch processing, stream processing requires continuous ingestion, transformation, and analysis of data in real-time. Apache Beam handles stream processing through **unbounded `PCollections`**, which represent datasets that may never be fully complete.

Examples of Stream Processing

- **Real-Time Fraud Detection:** Banks and financial institutions often use streaming data to detect fraudulent transactions in real-time, while analyzing transaction data as it flows in.
- **Real-Time Analytics:** Online platforms like e-commerce websites and social media apps use stream processing to generate real-time analytics for user interactions.
- **IoT Data Processing:** IoT devices constantly generate data (for example, temperature readings, location updates). Stream processing allows this data to be processed as it arrives.

Example of a Stream Processing Pipeline

Let us build a simple streaming pipeline that monitors incoming temperature sensor data and alerts when the temperature crosses a threshold.

```
import apache_beam as beam

# Function to check for high temperatures
def check_high_temperature(reading):
 timestamp, temperature = reading.split(',')
 if float(temperature) > ٧٥.٠:
  return True
 return False

p = beam.Pipeline()

# Simulate a streaming data source (for example, from IoT
sensors)
```

```
temperature_data = p | 'ReadTemperatureStream' >>
beam.io.ReadFromPubSub(topic='projects/my-
project/topics/temperature')

# Filter readings where the temperature exceeds 75.0 degrees
high_temp_readings = temperature_data | 'FilterHighTemp' >>
beam.Filter(check_high_temperature)

# Write the high temperature readings to an alert system
high_temp_readings | 'WriteToAlerts' >>
beam.io.WriteToText('high_temp_alerts.txt')

p.run()
```

In this example:

- **Unbounded Data:** We are reading from a continuous stream of temperature sensor readings (an unbounded `PCollection`).
- **Real-Time Filtering**: Each incoming reading is checked, and if the temperature exceeds 75 degrees, it triggers an alert.
- **Continuous Output:** The pipeline continuously writes alerts to an output sink as high temperatures are detected, without waiting for the entire stream to be processed (since it is unbounded).

Thus, stream processing pipelines like this one are used in situations where time is critical and real-time insights are necessary.

Unified API for Batch and Streaming Processing

The true power of Apache Beam lies in its **unified API**, which allows you to use the same pipeline code for both batch and streaming scenarios. The way Beam abstracts the data as a `PCollection`—whether it is bounded or unbounded—enables the same logic to be applied to both static and dynamic datasets.

- **Bounded `PCollection` (Batch):** Represents finite datasets where processing occurs after all the data is available
- **Unbounded `PCollection` (Stream):** Represents infinite datasets where processing happens in real-time as data arrives

This unified API means you can design a pipeline once, and depending on your data source, it can either be a batch job or a streaming job without requiring you to make significant changes to your code.

Here is a breakdown of how Beam achieves this unification:

- **Same Code, Different Sources:** When you write a Beam pipeline, you do not need to worry about whether the data is bounded or unbounded. The runner will adapt the execution based on the source of the data. For example, the same transformation logic can be applied to both a batch of log files (bounded) and a real-time stream of log entries (unbounded).
- **Windowing and Triggers**: In stream processing, since data is unbounded, you often want to group it into windows of time (for example, every 5 minutes, hourly, daily) to apply transformations over manageable segments of data. Windowing becomes mandatory when working with **unbounded `PCollections`**, because the data stream is infinite and cannot be processed as a complete dataset; windows provide finite, logical boundaries that make it possible to apply aggregations, grouping, and other stateful computations over streaming data. Apache Beam offers rich support for windowing, which allows you to group unbounded data into logical units for processing.
    - **Fixed Windows:** Divide data into fixed-duration windows (for example, 1-minute windows).
    - **Sliding Windows:** Create overlapping windows that slide over time (for example, every 30 seconds, create a 1-minute window).
    - **Session Windows:** Group events into sessions based on periods of activity separated by gaps of inactivity.

Example of a Windowed Streaming Pipeline

```
import apache_beam as beam
from apache_beam.transforms.window import FixedWindows

# Pipeline that processes streaming sensor data with fixed
windowing
p = beam.Pipeline()

# Ingest data from a streaming source
```

```
sensor_data = p | 'ReadSensorData' >>
beam.io.ReadFromPubSub(topic='projects/my-
project/topics/sensor')

# Apply fixed windowing (for example, 5-minute windows)
windowed_data = sensor_data | 'ApplyWindow' >>
beam.WindowInto(FixedWindows(300))

# Perform some transformation (for example, calculate average
reading per window)
avg_readings = windowed_data | 'Average' >>
beam.CombineGlobally(beam.combiners.MeanCombineFn()).without_d
efaults()
# Write the results to an output sink
avg_readings | 'WriteToStorage' >>
beam.io.WriteToText('average_readings.txt')

p.run()
```

- **Triggers:** In streaming pipelines, you may want to control when the results of windowed computations are emitted. This is done using **triggers**. Triggers define when the results should be output, and they are particularly useful when dealing with unbounded data.

  Common trigger types include:

  - **Event Time triggers:** They emit results based on event timestamps.
  - **Processing Time triggers:** They emit results based on system clock time.
  - **Count-Based triggers:** They emit results after a certain number of elements.

Triggers allow you to balance between latency (how quickly you get results) and completeness (how much data you wait for before processing).

Aspect	Batch Processing	Stream Processing
Data Type	Bounded (finite)	Unbounded (infinite)
Processing Model	Processes entire dataset at once	Processes data as it arrives in real-time
Windowing	Not required	Required to group continuous data

Example Use Cases	ETL jobs, historical reporting	Real-time analytics, IoT data processing
Latency	High (results are produced after data is available)	Low (results are produced continuously)
Common Technologies	Apache Hadoop, Apache Spark (batch mode)	Apache Flink, Apache Kafka Streams

Table 1.3: *Batch vs. Streaming- Key Differences*

Apache Beam abstracts away these differences by treating both batch and streaming data as `PCollections`, and applying the same pipeline logic. This allows developers to focus on business transformations, while the runner handles whether the data is bounded or unbounded.

`PCollection` and `PTransform`: Building Blocks of Beam

In Apache Beam, `PCollection` and `PTransform` are the core building blocks for constructing data pipelines. `PCollections` represent data, and `PTransforms` define the operations applied to this data. These abstractions allow Beam to handle data processing in a consistent and flexible manner, regardless of whether the data is bounded or unbounded, batch or stream.

Therefore, in this section, we will explore both concepts in detail, explaining how they work, how they interact, and how they are used in pipeline development.

`PCollection`: The Fundamental Data Structure

A `PCollection` is the central data abstraction in Apache Beam. It represents a distributed dataset that can either be **bounded** (for batch processing) or **unbounded** (for streaming). In practice, a `PCollection` is much like a collection in traditional programming languages, except that it is designed to be distributed across multiple machines and processed in parallel.

Characteristics of `PCollection`:

- **Bounded vs. Unbounded Data**
    - **Bounded `PCollection`**: A bounded `PCollection` represents a finite dataset where all the data is available from the start. These

are typically used in batch processing jobs, where data is static and known in advance (for example, processing a file stored on disk).

- **Unbounded** `PCollection`: An unbounded `PCollection` represents an infinite stream of data. Unbounded `PCollections` are typically used in streaming jobs, where data continuously flows in, and it must be processed as it arrives (for example, real-time logs or sensor data streams).

- **Distributed and Parallel Processing:** `PCollections` are distributed across a cluster, meaning that each element in a `PCollection` can be processed independently in parallel. This makes Beam highly scalable, as it allows pipelines to handle large volumes of data efficiently.
- **Element Types:** A `PCollection` can contain elements of any type (for example, strings, integers, key-value pairs, or complex data structures). These elements are processed individually by Beam transformations.
- **Immutability:** `PCollections` are immutable, which means that once data is in a `PCollection`, it cannot be changed. Any transformation applied to a `PCollection` results in a new `PCollection`, while ensuring a functional and parallelizable approach to data processing.

Creating `PCollections`: `PCollections` can be created from a variety of sources, including reading from text files, databases, message queues, or even in-memory data. Beam provides built-in I/O transforms to ingest data from various sources, and then turn them into `PCollections`.

Example: Creating a `PCollection` from a List of Elements

```
import apache_beam as beam

# Create a simple in-memory PCollection
with beam.Pipeline() as p:
 numbers = p | ‹Create numbers› >> beam.Create([1, 2, 3, 4,
 5])
```

In this example, the `Create` transform is used to manually create a `PCollection` from a list of numbers.

Example: Creating a `PCollection` by Reading from a File

```
with beam.Pipeline() as p:
```

```
lines = p | ‹ReadFile' >> beam.io.ReadFromText('input.txt')
```

In this case, the **`ReadFromText`** transform reads lines from a text file, and stores them as elements in a **`PCollection`**.

`PTransform`: Operations on `PCollections`

A **`PTransform`** is the operation that you apply to a **`PCollection`**. These transforms are the building blocks that allow you to process data in Apache Beam. **`PTransforms`** take one or more **`PCollections`** as input, apply a transformation to the data, and produce one or more **`PCollections`** as output.

Types of `PTransforms`

- **Element-Wise Transforms:** These transforms operate on each element of the input **`PCollection`** individually. Examples of element-wise transforms include **`Map`**, **`FlatMap`**, and **`Filter`**. These are explained as follows:
  - **`Map`**: It applies a function to each element in the **`PCollection`**, and returns a new **`PCollection`** with the transformed elements.
  - **`FlatMap`**: This is similar to `Map`, but allows the function to return multiple elements for each input element.
  - **`Filter`**: It removes elements from the **`PCollection`** that do not meet a specified condition.

  Example of `Map`:

  ```
  with beam.Pipeline() as p:
    numbers = p | ‹Create numbers› >> beam.Create([1, 2, 3, 4, 5])
    squares = numbers | ‹Square numbers› >> beam.Map(lambda x: x * x)
  ```

  In this example, the **`Map`** transform applies a lambda function to generate a square of each number in the **`PCollection`**.

  Example of `FlatMap`:

  ```
  with beam.Pipeline() as p:
  ```

```
lines = p | ‹ReadFile' >>
beam.io.ReadFromText('input.txt')
words = lines | ‹SplitWords' >> beam.FlatMap(lambda
line: line.split())
```

Here, **`FlatMap`** splits each line of text into words and produces multiple elements for each input element.

- **Key-Based Transforms:** Key-based transforms operate on **key-value pairs** within a **`PCollection`**. These transforms are used for grouping, joining, and aggregating data. Examples include **`GroupByKey`**, **`CombinePerKey`**, and **`CoGroupByKey`**.
    - **`GroupByKey`**: It groups elements in a **`PCollection`** by key.
    - **`CombinePerKey`**: It applies an aggregation function (such as sum or average) to the values grouped by key.

Example of `GroupByKey`:

```
with beam.Pipeline() as p:
  pairs = p | 'Create pairs' >> beam.Create([('apple', 1),
  ('banana', 2), ('apple', 3)])
  grouped = pairs | 'Group by key' >> beam.GroupByKey()
```

In this example, **`GroupByKey`** groups elements by their key (for example, all "**`apple`**" key-value pairs are grouped together).

- **Aggregating Transforms**: Aggregating transforms combine data across the entire **`PCollection`**, or within key groups. Examples include **`Sum`**, **`Count`**, and **`Combine`.**
    - **`Sum`:** Adds up all elements in the **`PCollection`**
    - **`Count`:** Counts the number of elements in the **`PCollection`**
    - **`Combine`:** Applies a custom function to aggregate the elements

Example of Count:

```
with beam.Pipeline() as p:
 elements = p | ‹Create elements› >> beam.Create([1, 2, 3,
 4, 5])
 count = elements | ‹Count elements› >>
 beam.combiners.Count.Globally()
```

This pipeline counts the total number of elements in the **`PCollection`**.

- **Composite Transforms:** Composite transforms are a combination of multiple **`PTransforms`** into a single reusable operation. They allow you to encapsulate complex processing logic into a single unit, while improving code readability and reusability.

Example of a Custom Composite Transform:

```
class FilterAndSquare(beam.PTransform):
  def expand(self, pcoll):
    return (pcoll
        | ‹Filter Even Numbers› >> beam.Filter(lambda x: x % 2
        == 0)
        | ‹Square Numbers› >> beam.Map(lambda x: x * x))

with beam.Pipeline() as p:
  numbers = p | ‹Create numbers› >> beam.Create([1, 2, 3, 4,
  5])
  result = numbers | ‹Filter and Square› >> FilterAndSquare()
```

In this example, **`FilterAndSquare`** is a composite transform that filters even numbers and squares them.

Applying Transforms to `PCollections`: The core idea behind Beam's data processing model is that you define a series of transforms that are applied to **`PCollections`**, which results in new **`PCollections`** at each step. This allows you to chain multiple operations together to build complex data pipelines.

Let us look at an extended example where we chain multiple transforms together:

Example: Processing a Text File to Count Words

```
import apache_beam as beam

with beam.Pipeline() as p:
  # Step ١: Read input from a text file
  lines = p | ‹ReadFile' >> beam.io.ReadFromText('input.txt')

  # Step ٢: Split each line into words
  words = lines | ‹SplitWords' >> beam.FlatMap(lambda line:
  line.split())

  # Step ٣: Assign each word a count of ١
```

```
word_pairs = words | 'PairWordsWithOne' >> beam.Map(lambda
word: (word, 1))

# Step ٤: Group words and sum their counts
word_counts = word_pairs | 'CountWords' >>
beam.CombinePerKey(sum)

# Step ٥: Write the results to an output file
word_counts | 'WriteResults' >>
beam.io.WriteToText('word_counts.txt')
```

In this pipeline:

1. **Reading Input**: Data is read from a text file using **`ReadFromText`**, which outputs a **`PCollection`** of lines.
2. **Transformations**: The lines are split into words (**`FlatMap`**), where each word is paired with the number 1 (**`Map`**), and the word counts are summed (**`CombinePerKey`**).
3. **Output**: The results are written to a text file using **`WriteToText`**.

Custom Transforms

One of the strengths of Apache Beam is its ability to let you define **custom transforms** to encapsulate reusable logic. Custom transforms are particularly useful when your pipeline has complex, repetitive logic that can be abstracted into a single unit.

Here is how you can create a custom transform in Beam:

Example: Custom Transform to Filter and Process Data

```
class FilterAndDouble(beam.PTransform):
 def expand(self, pcoll):
  return (pcoll
      | ‹Filter Even Numbers› >> beam.Filter(lambda x: x % 2
      == 0)
      | ‹Double Numbers› >> beam.Map(lambda x: x * 2))

with beam.Pipeline() as p:
 numbers = p | ‹Create numbers› >> beam.Create([1, 2, 3, 4, 5,
 6])
```

```
result = numbers | ‹Apply Filter and Double› >>
FilterAndDouble()
```

In this example:

The **`FilterAndDouble`** custom transform filters even numbers and doubles them. The pipeline, therefore becomes more modular, easier to maintain, and reusable.

Building Your First Apache Beam Pipeline

Now that we have covered the foundational components of Apache Beam, including **`PCollections`** and **`PTransforms`**, it is time to put those concepts into practice by building your first complete Apache Beam pipeline. Hence, in this section, we will walk through a step-by-step process of constructing a full pipeline, including data ingestion, transformation, and output. We will also explore how to run pipelines in different environments, whether locally or on a cloud-based runner.

Therefore, by the end of this section, you will understand how to design, execute, and manage a basic Beam pipeline, as well as how to deploy it on various execution engines (runners) such as **`DirectRunner`**, **`Google Cloud Dataflow`**, **`Apache Flink`**, or **`Apache Spark`**.

Step-by-Step Pipeline Construction

To guide you through the construction of a Beam pipeline, we will build a real-world example that processes data from a CSV file, transforms the data, and then outputs the results to a text file. The example demonstrates how to read data, apply multiple transformations, and write results, while explaining each stage of the pipeline.

Let us break it down into manageable steps.

Step 1: Set up the Pipeline Environment

Every Apache Beam pipeline starts by initializing the pipeline environment, which is done by creating a **`Pipeline`** object. This object acts as the container for all the transforms that will be applied to the data.

```
import apache_beam as beam

# Initialize the pipeline
```

```
with beam.Pipeline() as p:
  # Your pipeline logic will go here
```

When initializing a pipeline, you can also specify a **runner**. The runner determines where your pipeline will be executed. For local development and testing, the **`DirectRunner`** is typically used. For production workloads, you may use **`Google Cloud Dataflow`**, **`Apache Flink`**, or **`Apache Spark`**.

Example of Specifying a Runner:

```
with beam.Pipeline(runner='DirectRunner') as p:
  # Your pipeline logic
```

For more advanced use cases, you can configure your runner with additional options, such as project credentials for cloud runners.

Step 2: Data Ingestion

The first step in any data pipeline is to ingest the data. Apache Beam provides a wide range of connectors to read data from different sources, including files, databases, message queues, and cloud storage.

For this example, we will read data from a **CSV file**. Beam's **`ReadFromText`** transform allows us to read lines from a text file, and treat them as individual elements in a **`PCollection`**.

Example: Reading from a CSV File:

```
# Read data from a CSV file
lines = p | 'ReadFromCSV' >> beam.io.ReadFromText('data.csv')
```

Here, each line in the CSV file will be represented as a string in the resultant **`PCollection`**. However, before we proceed to transform the data, we need to split each line into individual fields.

Step 3: Transforming Data

Once the data is ingested, the next step is to apply transformations to clean, filter, or process it. Beam's **`PTransform`** functions like **`Map`**, **`FlatMap`**, **`Filter`**, and others are used to perform these operations.

Let us start by splitting each line from the CSV file into its individual components (for example, splitting the line based on commas).

Example: Splitting CSV Lines into Fields:

```
# Define a function to split each line into fields
def split_line(line):
```

```
  return line.split(',')

# Apply the transformation
split_data = lines | 'SplitLines' >> beam.Map(split_line)
```

The **`Map`** transform applies the **`split_line`** function to each element in the **`PCollection`**, while converting each string into a list of fields. For instance, a line like **`"1,John,Doe,25"`** will be transformed into **`['1', 'John', 'Doe', '25']`**.

Applying Multiple Transformations

Once the data is split, we can apply further transformations. For instance, we may want to filter out certain rows, or modify specific fields. Let us filter out rows where the age field is less than 30:

```
# Define a filtering function
def filter_by_age(fields):
  age = int(fields[٣])  # Assuming the ٤th field is age
  return age >= ٣٠

# Apply the filter transformation
filtered_data = split_data | 'FilterByAge' >>
beam.Filter(filter_by_age)
```

In this example, the `Filter` transform applies the **`filter_by_age`** function to each element in the **`PCollection`**, while removing rows where the age is less than 30.

Next, we can map the data to key-value pairs if needed. This is often useful for grouping or aggregating data. Let us convert the data into key-value pairs, where the key is the first field (for example, an ID), and the value is the rest of the fields.

Example: Mapping to Key-Value Pairs

```
# Define a mapping function
def to_key_value(fields):
  return (fields[٠], fields[١:])  # ID as key, rest as value

# Apply the map transformation
key_value_data = filtered_data | 'ToKeyValue' >>
beam.Map(to_key_value)
```

Now, we have a **`PCollection`** of key-value pairs, where the key is an ID and the value is the rest of the fields.

Step 4: Write the Data

Once the data has been transformed, the next step is to output the results. Apache Beam supports a variety of output connectors, including file systems, databases, and cloud services. Hence, for this example, we will write the output to a text file.

Example: Writing to a Text File

```
# Write the processed data to an output file
key_value_data | 'WriteToFile' >>
beam.io.WriteToText('output.txt')
```

This will write each element in the **`PCollection`** to an output file, where each line in the file corresponds to one element in the **`PCollection`**.

Step 5: Run the Pipeline.

After defining the pipeline steps, the next task is to run the pipeline. When you invoke **`p.run()`**, Beam processes the data according to the pipeline's transforms. If you are using the **`DirectRunner`**, the execution happens locally.

```
# Run the pipeline
p.run()
```

If you are using a cloud-based runner, such as **`Google Cloud Dataflow`**, the pipeline will be executed on a distributed cluster, and you may need to provide additional configuration options, such as project credentials and cloud storage locations.

Building a More Complex Pipeline

Now that we have walked through the basics, let us extend this pipeline to include some more advanced features, such as grouping and aggregation. Suppose, we want to count how many times each unique name appears in the dataset.

- **`GroupByKey`**: We will use the **`GroupByKey`** transform to group all entries by name.
- **`CombinePerKey`**: Then, we will apply **`CombinePerKey`** to count the occurrences of each name.

Example: Counting Occurrences of Names

```
# Map names to key-value pairs where the value is 1
def to_name_count(fields):
 name = fields[\]  # Assuming the name is in the second field
 return (name, \)

# Apply the mapping
name_counts = split_data | 'MapNamesToCount' >>
beam.Map(to_name_count)

# Group by name and sum the counts
name_totals = name_counts | 'CountPerName' >>
beam.CombinePerKey(sum)

# Write the results to a file
name_totals | 'WriteNameCounts' >>
beam.io.WriteToText('name_counts.txt')
```

In this extended example:

- We first map the names to key-value pairs, where the key is the name, and the value is the count (initially 1).
- We then group by name using **`CombinePerKey`**, while summing up the counts for each unique name.
- Finally, the results are written to a file.

Hence, this example showcases how Beam allows you to easily group and aggregate data at scale.

Running the Pipeline on Different Runners

While developing and testing pipelines locally with the **`DirectRunner`** is convenient, you will eventually want to run your pipeline on a distributed execution engine, such as Google Cloud Dataflow, Apache Flink, or Apache Spark. The runner-specific setup and its limitations will be covered in more depth in *Chapter 7: Deploying Apache Beam Pipelines on Different Runners*. Meanwhile, running the pipeline can look like this on different runners:

- **Running Locally with `DirectRunner`**

  `DirectRunner` is ideal for small datasets and local testing. This runner executes the pipeline on your local machine, which allows you to

quickly iterate and debug the pipeline.

- **Running on `Google Cloud Dataflow`**

  For large-scale production pipelines, **`Google Cloud Dataflow`** provides a fully managed, scalable service. To use Dataflow, you need to configure additional options, such as the project ID and output paths.

Example of Configuring `Dataflow Runner`:

```
options = beam.options.pipeline_options.PipelineOptions(
 runner=›DataflowRunner›,
 project=›my-project-id›,
 temp_location='gs://my-bucket/temp'
)

with beam.Pipeline(options=options) as p:
 # Your pipeline logic
```

- **Running on `Apache Flink`**

  `Apache Flink` is another popular distributed execution engine for Beam pipelines. To run on Flink, you will need to configure the pipeline for **`FlinkRunner`** and deploy it to a Flink cluster.

  Example:

  ```
  options = beam.options.pipeline_options.PipelineOptions(
   runner=›FlinkRunner'

  with beam.Pipeline(options=options) as p:
   # Your pipeline logic
  ```

- **Running on `Apache Spark`**

  Apache Spark can also be used to run Beam pipelines in distributed environments. The process is like configuring for Flink or Dataflow.

Optimizing and Debugging Pipelines

Once your pipeline is running, it is important to monitor its performance and optimize it for efficiency. Beam provides various tools and strategies to help you with this, which are explained as follows:

- **Pipeline Monitoring:**

Cloud-based runners such as Google Cloud Dataflow provide detailed monitoring and logging for each stage of the pipeline, which helps you track job progress, view error logs, and optimize resource usage.

- **Fusion Optimization:**

  Beam automatically optimizes pipelines by fusing adjacent operations into more efficient execution units. This reduces the overhead of moving data between steps, and improves pipeline performance.

- **Windowing for Streaming Pipelines:**

  When working with streaming data, you can use windowing strategies to manage unbounded datasets, grouping data into fixed-size windows for processing. This allows you to control how data is aggregated over time, and also ensures that your streaming pipelines produce timely results.

Conclusion

In this introductory chapter, we have covered the foundational concepts that underpin Apache Beam, while providing a comprehensive and clear understanding of how Beam serves as a unified model for both batch and streaming data processing. Through the lens of key components like `PCollection` and `PTransform`, we have explored how Beam abstracts data and operations. These components make Beam a versatile solution for processing bounded and unbounded datasets alike.

We also built a full Beam pipeline step by step, which shows how to:

- Ingest data from various sources.
- Apply transformations using Beam's rich library of transforms.
- Output results to different destinations.
- Execute pipelines locally or on distributed systems such as Google Cloud Dataflow and Apache Flink.

Hence, the power of Apache Beam lies in its ability to handle both batch and streaming data with the same code, while offering flexibility, scalability, and portability across multiple execution engines (runners). Whether you are working with finite data sets (batch), or processing real-

time streams, Beam provides the tools to handle these scenarios efficiently and seamlessly.

In *Chapter 2: Stateful and Stateless Processing with Apache Beam*, we will build on these foundational concepts by exploring **stateful and stateless processing in Apache Beam**. You will learn how Beam manages state across streaming pipelines, how stateful operations enable more advanced use cases such as session tracking, anomaly detection, and complex event processing, and when simple stateless transformations are sufficient. Thus, the progression will deepen your understanding of how Beam handles real-world streaming scenarios where context, memory, and time play a critical role in producing accurate and meaningful results.

References

- **Google Cloud**: Apache Beam & Cloud Dataflow Customer Stories: https://cloud.google.com/dataflow/docs/resources/customer-stories
- **Forrester**: The Total Economic Impact™ of Real-Time Data Platforms: https://www.forrester.com/report/the-total-economic-impact-of-real-time-data-platforms/

You've Just Finished your Free Sample

Enjoyed the preview?

Buy: http://www.ebooks2go.com