**Abstract**

Streaming data is the data generated continuously from different sources. The river has no beginning and no end. Likewise, streaming data is ideally suited to data that has no discrete beginning or end. Stream Processing is a part of Big Data Analytics which aims at querying the continuous data stream and extracting useful and meaningful data from the stream. Google’s MapReduce paper in 2005 introduced simple and scalable data processing techniques. Most initial use cases earlier can be solved using batch analytics but nowadays there are applications like stock market, patient monitoring and traffic analysis which can cause a drastic difference if the output is generated in levels of hours and minutes. For an example, stock markets analytics results are often useless within milliseconds. Batch system outputs are generally in terms of minutes and hours. In such use cases batch system processing fails. So here comes the requirement of stream analytics. Big data is bringing in an era of revolution across sectors and financial markets are no exemption. Big Data will be the most dependable tool for accurate and informed decision-making in financial trading.

Stock market has been an area of intense interest due to its high return on invested money in short span of time. The stock price coming from transactions are real-time, continuous, ever changing with time. This continuous variation in the prices lays the customer bewildered.

This attempt tries to predict the close value of share for specific company. The prediction varies on the current value, low, high and open value of the share.

**Chapter 1 Introduction**

**1.1 Overview**

Stock market prediction is the act of trying to determine the future value of a company stock or other financial instrument traded on a financial exchange. The successful prediction of a stock's future price will maximize investor’s gains. Predicting how the stock market will perform is one of the most difficult things to do. There are so many factors involved in the prediction – physical factors, psychological, rational and irrational behavior etc. All these aspects combine to make share prices volatile and very difficult to predict with a high degree of accuracy. Stock market analysis and prediction are being studied using various methods such as machine learning and text mining. Data mining studies usually daily stock data. These studies predict daily stock prices using the daily closing price, which is not sufficient to make predictions in a short period of time (e.g., 1 hour or 30 minutes).

In this project there is an attempt to continuously predict the next probable close value of the share using the streaming analytics of the big data. In this project we are fetching the data by querying from the website. After fetching, the data is given as input to apache beam pipeline which has various transformations in it and is executed on Google cloud dataflow. The Regression model of machine learning is integrated for the stock market share price prediction.

The analytical comparison on exactly once and at least once behavior of stream processing engine has also been reviewed on.

**1.2 Problem definition**

To study the sea changes arising from the advances in real time streaming data field.  Stock market is very vast and difficult to understand. It is considered too uncertain to be predictable due to huge fluctuation of the market. Stock market prediction task is interesting as well as challenging. Investing in a good stock but at a bad time can have disastrous result, while investing in a stock at the right time can bear profits. Financial investors of today are facing this problem of trading as they do not properly understand. The predictions being done till now where on the basis of past data and in the time interval of days. Here we are trying to predict the probable close value of the share every 30 seconds interval on the basis of Open, High, Low, Change, Current and many such parameters. There are some inherent challenges for stream-based analysis. Due to the continuous, unbounded high speed nature of the data streams, the amount of data processed is huge and there is not enough time to go through the data again and again iteratively. Also there is not enough space to store these unbounded data streams. Due to this, single pass analysis that can work with a small amount of memory are necessary. In some domains underlying data distribution of a stream can change as the time pass by and the analysis should be able to adapt to these changes in the streams, otherwise the concept-drifting problem can take place. The model should process data faster than they arrive to avoid shredding of data and data sampling. The challenge here lies in handling the latency of the system and to minimize the average processing delay using the windowing concept using the apache beam pipeline.

**1.3 Motivation**

*“Streaming Data Offers Insights Not Found Elsewhere”*

Currently, the world is creating 2.5 quintillion bytes of [data](http://www-01.ibm.com/software/data/bigdata/what-is-big-data.html) daily and this represents a unique opportunity for processing, analyzing and leveraging the information in useful ways. Machine learning and algorithms are increasingly being used in financial trading to compute vast quantities of data and make predictions and decisions that humans just do not have the capacity for. Real-time analytics has the potential to improve the investing power of the firms and individuals. Real Time Streaming Analysis is a need in today’s era for the companies as it is the requirement to instantly recognize the fraudulent behavior and take the necessary actions which sometimes go undetected or is too late to be detected in the normal systems which performs analysis on past data. Enterprises are exploring a variety of architectures and technologies to incorporate real-time analytics on streaming data into their ecosystems. Instead of the traditional approach of custom coding and integrations that apply in only limited situations, this new paradigm simplifies the extraction of value from streaming data. There are various applications of streaming Analytics like alerting responses, retail industry, cyber security, traffic handling and stock market predictions. This application motivate towards working in this cutting edge technology. There are many challenges theoretically as well as practically which motivates to work in this area.

**1.4 Scope & Objective of Research**

The main objective behind streaming data analysis is stock values fluctuate continuously whose effect cannot be visualized accurately in the prediction using historical data. So, use live streaming updates of the stock values in the process of prediction to be more accurate using concepts like handling late data latency and using windowing and triggering concepts.

The predictions being done till now is on the basis of past data and on the daily frequency basis. The model is predicting the close value every 30 seconds. The accuracy of our model is 90.25%.

**Chapter 2 Literature Review**

**2.1 Study of existing System**

* + 1. **Apache Hadoop**

**i) Introduction**

Apache Hadoop was born to enhance the usage and solve major issues of big data. The web media was generating loads of information on a daily basis, and it was becoming very difficult to manage the data of around one billion pages of content. In order of revolutionary, Google invented a new methodology of processing data popularly known as MapReduce.

Apache Hadoop is the most important framework for working with Big Data is based on MapReduce Technique for distributed computing. Hadoop’s biggest strength is scalability. It upgrades from working on a single node to thousands of nodes without any issue in a seamless manner. It is a framework which is based on [java programming](https://intellipaat.com/java-training/). It is intended to work upon from a single server to thousands of machines each offering local computation and storage. It supports the large collection of data set in a distributed computing environment.

The Apache Hadoop software library based framework that gives permissions to distribute huge amount of data sets processing across clusters of computers using easy programming models.

**ii) Architecture**

**Hadoop Architecture based on the following main components namely:**

1. **HDFS**

Hadoop Distributed File System provides unrestricted, high-speed access to the data application. It is based on the Google File System (GFS) which provides a distributed file system that is especially designed to run on commodity hardware. It reduces the faults or errors and helps incorporate low-cost hardware. It gives high level processing throughput access to application data and is suitable for [applications with large datasets](https://intellipaat.com/blog/big-data-analytics-tools-performance-testing/).

1. **MapReduce**

This is a highly efficient methodology for parallel processing of huge volumes of data. [It is a parallel programming model](https://intellipaat.com/tutorial/mapreduce-tutorial/introduction-of-mapreduce/) mainly used for writing large amount of data distribution applications devised from Google for efficient processing of large amounts of datasets, on large group of clusters.

1. **Hadoop Common**

Includes the common utilities which supports the other Hadoop modules.

1. **Hadoop YARN**

This technology is basically used for scheduling of job and efficient management of the cluster resource.

1. [**HBase**](https://intellipaat.com/hbase-training/)

It is the Hadoop file system and works as common storage for any type of Hadoop application. It is a non-relational, distributed database management system that works efficiently on sparse data sets and it is highly scalable.

1. **Hive**

It is a data warehouse tool basically used for analyzing, querying and summarizing of analyzed data concepts on top of the Hadoop framework.

1. **Pig**

Pig is a high-level framework which ensures us to work in coordination either with Apache Spark or MapReduce to analyze the data.

1. **Zookeeper**

Open source centralized service which is used to provide coordination between distributed applications of Hadoop. It offers the registry and synchronization service on a high level. It maintains information, naming, providing distribution synchronization, etc.

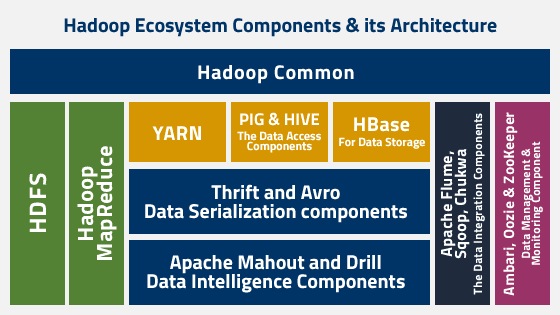


Fig. 2.1 Apache Hadoop Architecture

**2.1.2 Apache Storm**

**i) Introduction**

Apache Storm is a distributed real-time big data-processing system. Storm is designed to process vast amount of data in a fault-tolerant and horizontal scalable method. It is a streaming data framework that has the capability of highest ingestion rates. Though Storm is stateless, it manages distributed environment and cluster state via Apache ZooKeeper. It is simple and you can execute all kinds of manipulations on real-time data in parallel. Storm is easy to setup, operate and it guarantees that every message will be processed through the topology at least once.

**ii) Architecture**

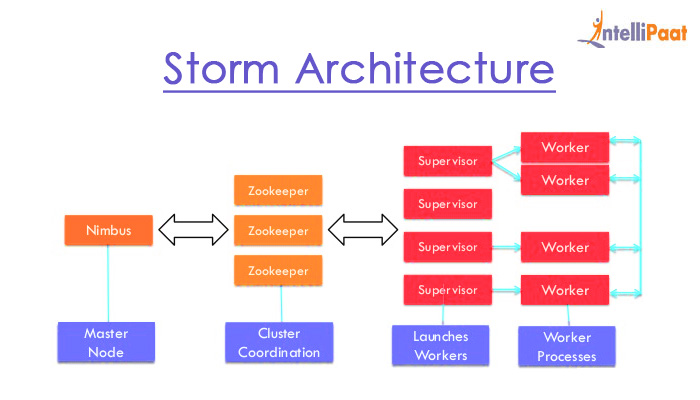


Fig. 2.2 Apache Storm Architecture

1. **Nimbus**

Nimbus is a master node of Storm cluster. All other nodes in the cluster are called as worker nodes. Master node is responsible for distributing data among all the worker nodes, assign tasks to worker nodes and monitoring failures.

1. **Supervisor**

The nodes that follow instructions given by the nimbus are called as Supervisors. A supervisor has multiple worker processes and it governs worker processes to complete the tasks assigned by the nimbus.

1. **Worker process**

A worker process will execute tasks related to a specific topology. A worker process will not run a task by itself instead it creates executors and asks them to perform a particular task. A worker process will have multiple executors.

1. **Executor**

An executor is nothing but a single thread spawns by a worker process. An executor runs one or more tasks but only for a specific spout or bolt.

1. **ZooKeeper**

Apache ZooKeeper is a service used by a cluster (group of nodes) to coordinate between themselves and maintaining shared data with robust synchronization techniques. Nimbus is stateless, so it depends on ZooKeeper to monitor the working node status. ZooKeeper helps the supervisor to interact with the nimbus. It is responsible to maintain the state of nimbus and supervisor.

* + 1. **Apache Spark**

**i) Introduction**

Apache Spark is a lightning fast cluster computing system. It provides the set of high-level API namely Java, Scala, Python, and R for application development. Apache Spark is a tool for speedily executing Spark Applications. Spark utilizes Hadoop in two different ways – one is for Storage and second is for Process handling. Just because Spark has its own Cluster Management, so it utilizes Hadoop for Storage objective.

Spark is intended to cover an extensive variety of remaining loads, for example, cluster applications, iterative calculations, intuitive questions, and streaming. Aside from supporting all these remaining tasks at hand in a particular framework, it decreases the administration weight of keeping up isolated apparatuses.

**ii) Architecture**

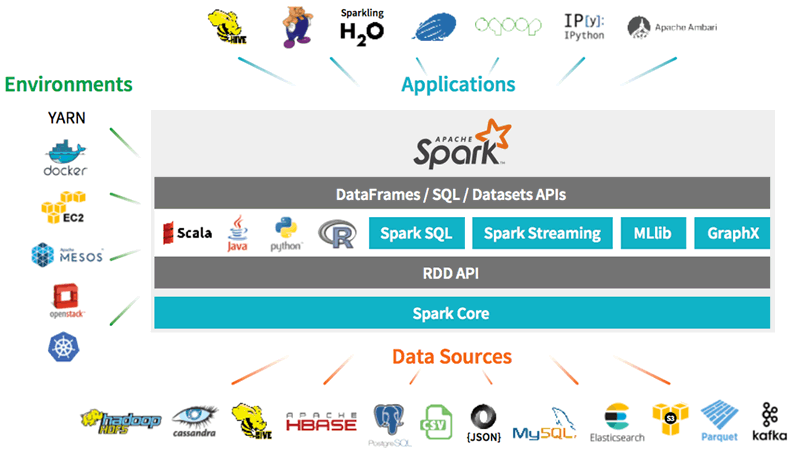


Fig. 2.3 Spark Architecture

1. **Apache Spark Core**   
   Spark Core is the basic general execution engine for the Spark platform that all other functionalities are based upon. It gives In-Memory registering and connected datasets in external storage frameworks.
2. **Spark SQL**

Spark SQL is a segment over Spark Core that presents another information abstraction called Schema RDD, which offers help for syncing structured and unstructured information.

1. **Spark Streaming**

Spark Streaming use Spark Core's quick scheduling ability to perform Streaming Analytics. It ingests information in scaled-down clusters and performs RDD (Resilient Distributed Datasets) changes on those small-scale groups of information.

1. **MLlib (Machine Learning Library)**

MLlib is a Distributed Machine Learning structure above Spark in view of the distributed memory-based Spark architecture. It is, as indicated by benchmarks, done by the MLlib engineers against the Alternating Least Squares (ALS) executions. Spark MLlib is nine times as rapid as the Hadoop disk version of Apache Mahout (before Mahout picked up a Spark interface).

1. **GraphX**

GraphX is a distributed Graph-Processing framework of Spark. It gives an API for communicating chart calculation that can display the client characterized diagrams by utilizing Pregel abstraction API. It likewise gives an optimized and improved runtime to this abstraction.

1. **Spark R**

It is R bundle that gives light-weight frontend. It permits running employments intuitively on them from the R shell. It incorporate the ease of use of R with the adaptability of Spark.

* + 1. **Apache Flink**

**i) Introduction**

The key vision for Apache Flink is to overcome and reduces the complexity that has been faced by other distributed data-driven engines. It is achieved by integrating query optimization, concepts from database systems and efficient parallel in-memory and out-of-core algorithms, with the[MapReduce](http://data-flair.training/blogs/hadoop-mapreduce-introduction-tutorial-comprehensive-guide/) framework. As Apache Flink is mainly based on the streaming model, Apache Flink iterates data by using streaming architecture. The concept of an iterative algorithm is tightly bounded into Flink query optimizer. Apache Flink’s pipelined architecture allows processing the streaming data faster with lower latency than micro-batch architectures ([Spark](http://data-flair.training/blogs/apache-spark-tutorial-quickstart-introduction/)).

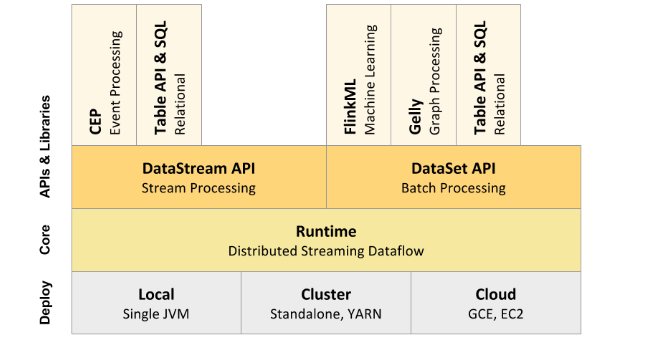


Fig. 2.3 Flink Architecture

1. **CEP(Complex event processing)**

It is the Complex Event Processing (CEP) library implemented on top of Flink. It allows you to detect event patterns in an endless stream of events, giving you the opportunity to get hold of what’s important in your data. It contains the pattern api that allows us to recognize individual patterns, combine the patterns and then grouping of the relevant patterns.

1. **Table API & SQL**

Apache Flink features two relational APIs - the Table API and SQL - for unified stream and batch processing. The Table API is a language-integrated query API for Scala and Java that allows the composition of queries from relational operators such as selection, filter, and join in a very intuitive way. Flink’s SQL support is based on [Apache Calcite](https://calcite.apache.org/) which implements the SQL standard. Queries specified in either interface has the same semantics and specify the same result regardless whether the input is a batch input (DataSet) or a stream input (DataStream).

1. **FlinkML**

FlinkML is the Machine Learning (ML) library for Flink. It is a new effort in the Flink community, with a growing list of algorithms and contributors. It supports supervised learning algorithms (SVM, Multiple Linear Regression), Unsupervised learning(k nearest neighbor join), data preprocessing (minmax scalar, standard scalar), outlier detection and so on.

1. **Gelly: Flink Graph API**

Gelly is a Graph API for Flink. It contains a set of methods and utilities which aim to simplify the development of graph analysis applications in Flink. In Gelly, graphs can be transformed and modified using high-level functions similar to the ones provided by the batch processing API. Gelly provides methods to create, transform and modify graphs, as well as a library of graph algorithms.

1. **DataStream API(Stream Processing)**

DataStream programs in Flink are regular programs that implement transformations on data streams (e.g., filtering, updating state, defining windows, aggregating). The data streams are initially created from various sources. Results are returned via sinks, which may for example write the data to files, or to standard output (for example the command line terminal). Flink programs run in a variety of contexts, standalone, or embedded in other programs. The execution can happen in a local JVM, or on clusters of many machines.

1. **Dataset API(Batch Processing)**

DataSet programs in Flink are regular programs that implement transformations on data sets (e.g., filtering, mapping, joining, grouping). The data sets are initially created from certain sources (e.g., by reading files, or from local collections). Results are returned via sinks, which may for example write the data to (distributed) files, or to standard output (for example the command line terminal). Flink programs run in a variety of contexts, standalone, or embedded in other programs. The execution can happen in a local JVM, or on clusters of many machines.

1. **Distributed Streaming Dataflow**

For distributed execution, Flink chains operator subtasks together into tasks. Each task is executed by one thread. Chaining operators together into tasks is a useful optimization: it reduces the overhead of thread-to-thread handover and buffering, and increases overall throughput while decreasing latency.

**2.2 Review of Literature & Findings**

Real Time Streaming Analysis is a need in today’s era for the companies as it is the requirement to instantly recognize the fraudulent behavior and take the necessary actions which sometimes go undetected or is too late to be detected in the normal systems which performs analysis on past data. There are various platforms for analyzing streaming data.

Apache Beam incorporates the different functionalities provided separately by Google and Apache independently into a single platform as shown below.

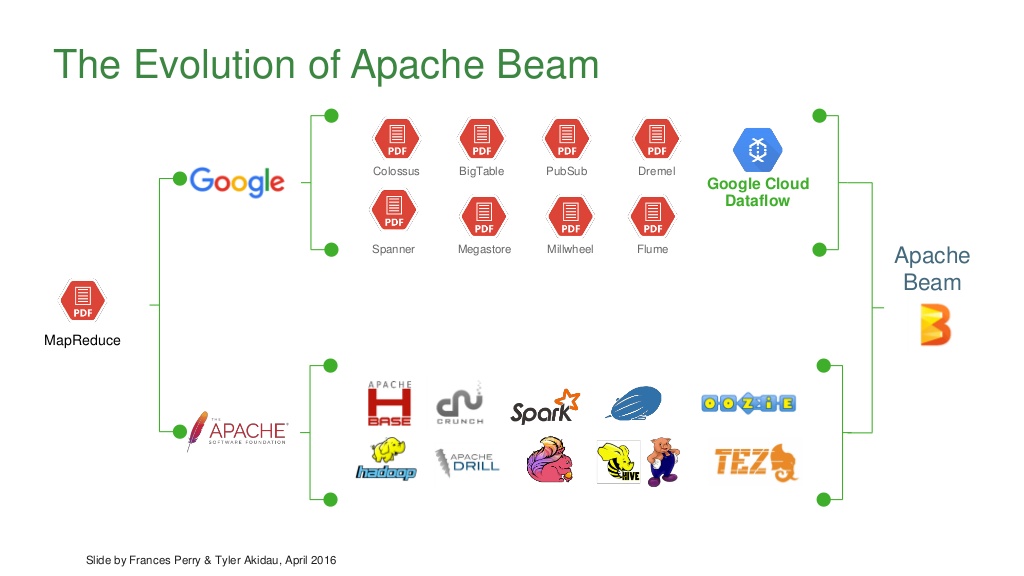


Fig. 2.4 Evaluation of Apache

Below description provides with the in detailed conclusion of the platforms.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Operations** | **Beam**  **Model** | **GCD** | **Apache Flink** | **Apache Spark** | **Hadoop**  **MapReduce** | **Apache Storm** |
| **ParDo** | Yes | Yes | Yes | Yes | Yes | Yes |
| **GBK** | Yes | Yes | Yes | Partial | Yes | Yes |
| **Flatten** | Yes | Yes | Yes | Yes | Yes | Yes |
| **Combine** | Yes | Yes | Yes | Yes | Yes | Yes |
| **Composite Transform** | Yes | Partial | Partial | Partial | Yes | Yes |
| **Side inputs** | Yes | Yes | Yes | Yes | Yes | Yes |
| **Splittable DoFn** | Partial | Yes | Yes | Partial | No | No |
| **Stateful Processing** | Yes | Partial | Partial | No | Partial | Partial |

Table 2.1 Comparison of Platforms

**2.2.1 Beam Model**

1. **ParDo:** It supports elementwise processing. Elements are processed in bundles, with initialization and termination hooks. Bundle size is chosen by the runner and cannot be controlled by user code. ParDo processes a main input PCollection one element at a time, but provides side input access to additional PCollections.
2. **GBK:** It does grouping of key-value pairs per key, window, and pane.
3. **Flatten:** It concatenates multiple homogenously typed collections together.
4. **Combine:** Application of an associative, commutative operation over all values (globally) or over all values associated with each key (per key).
5. **Composite Transforms:** It supports user-defined transformation subgraphs.
6. **Side inputs:** It supports additional elements available during DoFn execution.
7. **Splittable DoFn:** It supports partially splittable DoFn where processing of each element can be split for parallelism, or suspended and resumed.
8. **Stateful Processing:** It supports storage per key, per window

**2.2.2 GCD**

1. **ParDo:** It is fully supported. Batch mode uses large bundle sizes. Streaming uses smaller bundle sizes.
2. **GBK:** It is fully supported.
3. **Flatten:** It is fully supported.
4. **Combine:** It efficiently executes in the GCD
5. **Composite Transforms:** It is partially supported via inlining. Currently composite transformations are inlined during execution. The structure is later recreated from the names, but other transform level information (if added to the model) will be lost.
6. **Side inputs:** Batch mode supports a distributed implementation, but streaming mode may force some size restrictions. Neither mode is able to push lookups directly up into key-based sources.
7. **Splittable DoFn:** It does not yet support autotuning features of the Source API.
8. **Stateful Processing:** It partially supports non merging windows. SetState and MapState are not yet supported.

**2.2.3 Flink**

1. **ParDo:** It is fully supported is fully supported by Flink for both batch and streaming.
2. **GBK:** It is fully supported. It uses Flink's keyBy for key grouping. When grouping by window in streaming the Flink runner uses the Beam code. This guarantees support for all windowing and triggering mechanisms.
3. **Flatten:** It is fully supported.
4. **Combine:** It uses a combiner for pre-aggregation for batch and streaming.
5. **Composite Transforms:** It is partially supported via inlining.
6. **Side inputs:** Batch mode supports a distributed implementation, but streaming mode may force some size restrictions. Neither mode is able to push lookups directly up into key-based sources.
7. **Splittable DoFn:** Does not yet support autotuning features of the Source API.
8. **Stateful Processing:** It partially supports non merging windows.SetState and MapState are not yet supported.

**2.2.4 Spark**

1. **ParDo:** It applies per-element transformations is fully supported by Spark for both batch and streaming.
2. **GBK:** It is fully supported in Batch Mode. GroupByKey with multiple trigger firings in streaming mode is a work in progress.
3. **Flatten:** It is fully supported.
4. **Combine:** It uses Spark's combineByKeyand aggregatefunctions.
5. **Composite Transforms:** It is partially supported via inlining.
6. **Side inputs:** It uses Spark's broadcast variables. In streaming mode, side inputs may update but only between micro-batches.
7. **Splittable DoFn:** It Partially supports bounded-per-element SDFs
8. **Stateful Processing:** Yet to be implemented.

**2.2.5 Hadoop MapReduce**

1. **ParDo:** It is fully supported.
2. **GBK:** It is fully supported.
3. **Flatten:** It is fully supported.
4. **Combine:** It is fully supported.
5. **Composite Transforms:** It is fully supported.
6. **Side inputs:** It is fully supported.
7. **Splittable DoFn:**Yet to be implemented
8. **Stateful Processing:** State is supported for non-merging windows. SetState and MapState are not yet supported.

**2.2.6 Apache Storm**

1. **ParDo:** It is fully supported.
2. **GBK:** It is fully supported.
3. **Flatten:** It is fully supported.
4. **Combine:** It is fully supported.
5. **Composite Transforms:** It is fully supported.
6. **Side inputs:** It is fully supported.
7. **Splittable DoFn:** Yet to be implemented
8. **Stateful Processing:** State is supported for non-merging windows. SetState and MapState are not yet supported.

**Chapter 3 Analysis of Existing Work & Limitations**

**3.1 Working of Existing System**

**3.1.1 Apache Hadoop**

Hadoop does distributed processing for huge data sets across the cluster of commodity servers and works on multiple machines simultaneously. To process any data, the client submits data and program to Hadoop. HDFS stores the data while MapReduceprocess the data and Yarn divide the tasks.

**i) HDFS**

It has master-slave topology. It has got two daemons running, they are NameNode and DataNode.

1. **NameNode**

NameNode is the daemon running of the master machine. NameNode stores the directory tree of all files in the file system. It tracks where across the cluster the file data resides. It does not store the data contained in these files. When the client applications wants to add/copy/move/delete a file, they interact with NameNode. The NameNode responds to the request from client by returning a list of relevant DataNode servers where the data lives.

1. **DataNode**

DataNode daemon runs on the slave nodes. It stores data in the Hadoop File System. In functional file system data replicates across many DataNodes. On startup, a DataNode connects to the NameNode. It keeps on looking for the request from NameNode to access data. Once the NameNode provides the location of the data, client applications can talk directly to a DataNode, while replicating the data, DataNode instances can talk to each other.

**ii) MapReduce**

The general idea of the MapReduce algorithm is to process the data in parallel on your distributed cluster.

Hadoop MapReduce includes several stages:

1. In the first step, the program locates and reads the « input file » containing the raw data.
2. As the file format is arbitrary, there is a need to convert data into something the program can process. The « InputFormat » and « RecordReader » (RR) does this job. InputFormat uses InputSplit function to split the file into smaller pieces. Then the RecordReader transforms the raw data for processing by the map. It outputs a list of key-value pairs.
3. Once the mapper process these key-value pairs the result goes to « OutputCollector ». There is another function called « Reporter » which intimates the user when the mapping task finishes.
4. In the next step, the Reduce function performs its task on each key-value pair from the mapper.
5. Finally, OutputFormat organizes the key-value pairs from Reducer for writing it on HDFS.

**iii) Yarn**

It divides the task on resource management and job scheduling/monitoring into separate daemons. There is one ResourceManager and per-application ApplicationMaster.

The ResourceManger have two components – Scheduler and AppicationManager.

1. **Scheduler**

It only allocates resources to various competing applications. It does not restart the job after failure due to hardware or application failure. The scheduler allocates the resources based on an abstract notion of a container

1. **ApplicationManager**

Following are the tasks of ApplicationManager:-

* Accepts submission of jobs by client.
* Negotaites first container for specific ApplicationMaster.
* Restarts the container after application failure.

**3.1.2 Storm**

1. Initially, the nimbus will wait for the “Storm Topology” to be submitted to it.
2. Once a topology is submitted, it will process the topology and gather all the tasks that are to be carried out and the order in which the task is to be executed.
3. Then, the nimbus will evenly distribute the tasks to all the available supervisors.
4. At a particular time interval, all supervisors will send heartbeats to the nimbus to inform that they are still alive.
5. When a supervisor dies and doesn’t send a heartbeat to the nimbus, then the nimbus assigns the tasks to another supervisor.
6. When the nimbus itself dies, supervisors will work on the already assigned task without any issue.
7. Once all the tasks are completed, the supervisor will wait for a new task to come in.
8. In the meantime, the dead nimbus will be restarted automatically by service monitoring tools.
9. The restarted nimbus will continue from where it stopped. Similarly, the dead supervisor can also be restarted automatically. Since both the nimbus and the supervisor can be restarted automatically and both will continue as before, Storm is guaranteed to process all the task at least once.
10. Once all the topologies are processed, the nimbus waits for a new topology to arrive and similarly the supervisor waits for new tasks.

**3.1.3 Spark**

When the job enters the driver converts the code into a logical directed acyclic graph (DAG). Afterwards, the driver performs certain optimizations like pipelining transformations. Furthermore, it converts the DAG into physical execution plan with the set of stages. Meanwhile, it creates small execution units under each stage referred to as tasks. Then it collects all tasks and sends it to the cluster.

It is the driver program that talks to the cluster manager and negotiates for resources. After this cluster manager launches executors on behalf of the driver. At this point based on data, placement driver sends tasks to the cluster manager. Executors register themselves with the driver program before executors begin execution.

Executors execute all the tasks assigned by the driver. Meanwhile, the application is running, the driver program monitors the executors that run. In the spark architecture driver program schedules future tasks. All the tasks by tracking the location of cached data based on data placement. When it calls the stop method of sparkcontext, it terminates all executors. After that, it releases the resources from the cluster manager.

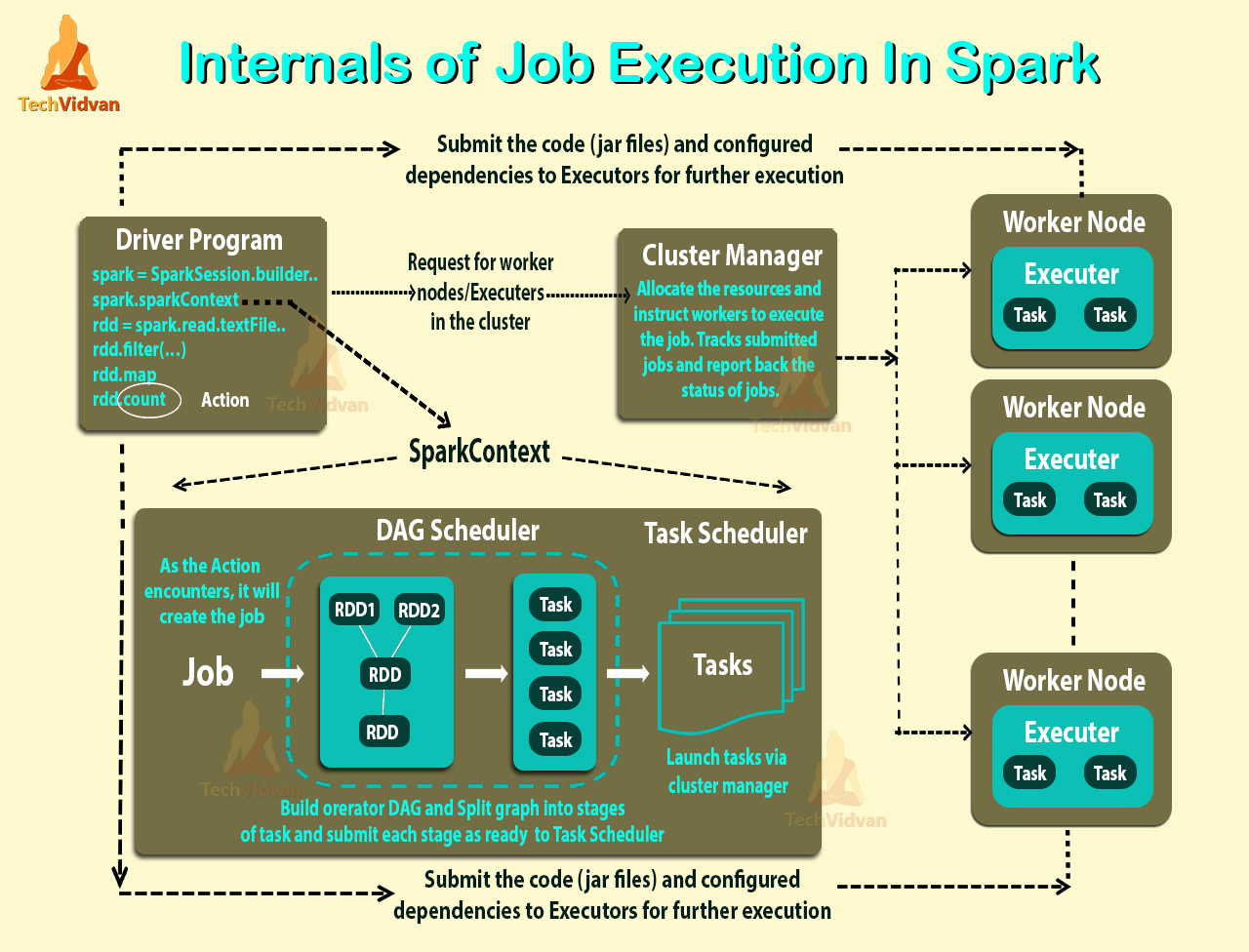


Fig. 3.1 Internal Working of Apache Spark

**3.2 Limitations of the Existing System**

**3.2.1 Limitations of Apache Hadoop**

1. Slow processing speed than spark
2. Support for batch processing only
3. No real-time data processing
4. High latency than spark
5. No abstraction
6. Uncertainty

**Solutions to all of the problem given by apache spark and apache flink**.

**3.2.2 Limitations of Apache Hadoop**

1. No real-time processing as the arriving live stream of data is divided into batches of pre-defined interval, and each batch of data is treated. So, gives nearly real-time processing not exactly real-time processing.
2. Spark MLib lacks behind in providing various functionalities
3. Manual optimization
4. High latency

**Chapter 4 Proposed Design**

**4.1 Introduction to Proposed System**

**4.1.1 Introduction**

The proposed system is built upon Apache Beam tool. Apache Beam is an open source from Apache Software Foundation. It is an unified programming model to define and execute data processing pipelines. The pipelines include ETL, batch and stream processing.

MapReduce has triggered the evolution of Big Data Ecosystem that we are seeing today. There are many frameworks like Hadoop, [Spark](https://www.tutorialkart.com/apache-spark-tutorial/), [Flink](https://www.tutorialkart.com/apache-flink-tutorial/" \t "_blank), Google Cloud Dataflow, etc, that came into existence. But there has been no unified API that binds all these frameworks and data sources, and provides an abstraction to the application logic from big data ecosystem. The good days have come, Apache Beam framework provides abstraction between your application logic and big data ecosystem.

Hence, there is no need to bother about the following aspects when writing data processing or analytic application:

1. **Data Source:** Data source can be batches, micro-batches or streaming data
2. **SDK:** You may choose your SDK ([Java](https://www.tutorialkart.com/java-tutorials/), Python) that you are comfortable with, to program application logic as a Beam Pipeline
3. **Runner:** Once writing of application logic as a Beam Pipeline is done, you may choose one of the available runners (Apache Spark, Apache Flink, Google Cloud Dataflow, Apache Apex, etc.) to run your application based on the nature of your inputs and analytic results.

Firstly we need to create a driver program using the classes in one of the Beam SDKs. Your driver program defines your pipeline, including all of the inputs, transforms, and outputs; it also sets execution options for your pipeline. These include the Pipeline Runner which in turn determines what back-end your pipeline will run on.

**4.1.2 Components of driver program**

**i) Pipeline**

A Pipeline encapsulates your entire data processing task, from start to finish. This includes reading input data, transforming that data, and writing output data. All Beam driver programs must create a Pipeline. When you create the Pipeline, you must also specify the execution options that tell the Pipeline where and how to run.

**ii) PCollection**

A PCollection represents a distributed data set that your Beam pipeline operates on. The data set can be bounded, meaning it comes from a fixed source like a file, or unbounded, meaning it comes from a continuously updating source via a subscription or other mechanism. Your pipeline typically creates an initial PCollection by reading data from an external data source, but you can also create a PCollection from in-memory data within your driver program. From there, PCollections are the inputs and outputs for each step in your pipeline.

**iii) PTransform**

A PTransform represents a data processing operation, or a step, in your pipeline. Every PTransform takes one or more PCollection objects as input, performs a processing function that you provide on the elements of that PCollection, and produces zero or more output PCollection objects.

**iv) I/O transforms**

Beam comes with a number of “IOs” - library PTransforms that read or write data to various external storage systems.

**4.1.3 Working of Beam driver program**

**i) Creating a pipeline**

Create a Pipeline object and set the pipeline execution options, including the Pipeline Runner. The Beam driver program typically starts by constructing a [Pipeline](https://github.com/apache/beam/blob/master/sdks/python/apache_beam/pipeline.py) object, and then using that object as the basis for creating the pipeline’s data sets as PCollections and its operations as Transforms. To use Beam, the driver program must first create an instance of the Beam SDK class Pipeline (typically in the main() function). When you create the Pipeline, you’ll also need to set some configuration options and pass them to the Pipeline object when you create the object.

**ii) PCollections**

It represents a potentially distributed, multi-element data set. It can be thought of a PCollection as “pipeline” data; Beam transforms use PCollection objects as inputs and outputs. If we want to work with data in our pipeline, it must be in the form of a PCollection. After we created the Pipeline, we will need to begin by creating at least one PCollection in some form. The PCollection we create serves as the input for the first operation in your pipeline.

### a) Creating a PCollection

You create a PCollection by either reading data from an external source using Beam’s Source API, or you can create a PCollection of data stored in an in-memory collection class in your driver program.

* Reading from an external source

To read from an external source, you use one of the [Beam-provided I/O adapters](https://beam.apache.org/documentation/programming-guide/#pipeline-io). The adapters vary in their exact usage, but all of them read from some external data source and return a PCollection whose elements represent the data records in that source.

Each data source adapter has a Read transform; to read, you must apply that transform to the Pipeline object itself. io.TextFileSource, for example, reads from an external text file and returns a PCollection whose elements are of type String, each String represents one line from the text file.

* Creating a PCollection from in-memory data

To create a PCollection from an in-memory list, you use the Beam provided Create transform.

**iii) Apply PTransforms to each PCollection**

Transforms can change, filter, group, analyze, or otherwise process the elements in a PCollection. A transform creates a new output PCollection without modifying the input collection. A typical pipeline applies subsequent transforms to each new output PCollection in turn until processing is complete. However, note that a pipeline does not have to be a single straight line of transforms applied one after another: think of PCollections as variables and PTransforms as functions applied to these variables.

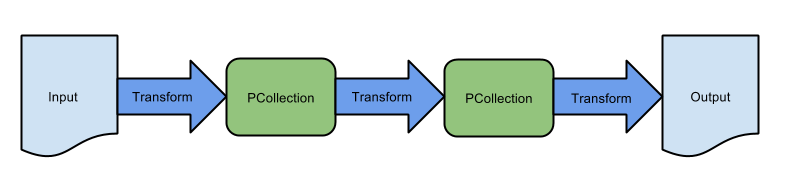


Fig. 4.1 A Linear Pipeline with Three Sequential Transforms

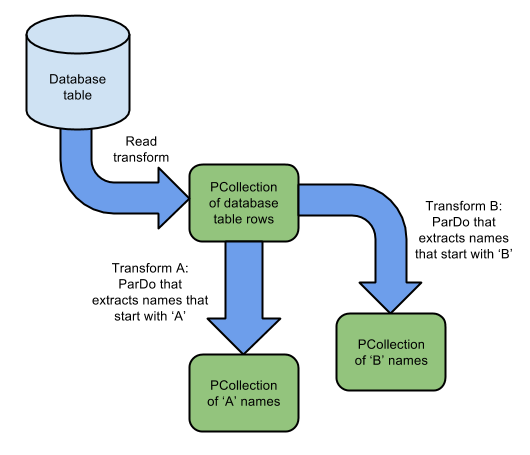


Fig. 4.2 A Branching Pipeline Showing Multiple Transforms

**Core Beam transforms**

Beam provides the following core transforms, each of which represents a different processing paradigm:

**a) ParDo**

It is a Beam transform for generic parallel processing. It considers each element in the input PCollection, performs some processing function (user code) on that element, and emits zero, one, or multiple elements to an output PCollection.

ParDo is useful for a variety of common data processing operations, including:

i) Filtering a data set.

ii) Formatting or type-converting each element in a data set.

iii) Extracting parts of each element in a data set.

iv) Performing computations on each element in a data set.

**b) GroupByKey**

It is a Beam transform for processing collections of key/value pairs. The input to GroupByKey is a collection of key/value pairs that represents a multimap, where the collection contains multiple pairs that have the same key, but different values. Given such a collection, you use GroupByKey to collect all of the values associated with each unique key.

GroupByKey is a good way to aggregate data that has something in common. For example, if you have a collection that stores records of customer orders, you might want to group together all the orders from the same postal code (wherein the “key” of the key/value pair is the postal code field, and the “value” is the remainder of the record).

**c) CoGroupByKey**

CoGroupByKey performs a relational join of two or more key/value PCollections that have the same key type.

Consider using CoGroupByKey if you have multiple data sets that provide information about related things. For example, let’s say you have two different files with user data: one file has names and email addresses; the other file has names and phone numbers. You can join those two data sets, using the user name as a common key and the other data as the associated values. After the join, you have one data set that contains all of the information (email addresses and phone numbers) associated with each name.

**d) Flatten**

Flatten is a Beam transform for PCollection objects that store the same data type. Flattenmerges multiple PCollection objects into a single logical PCollection.

**iv) Pipeline IO**

When you create a pipeline, you often need to read data from some external source, such as a file or a database. Likewise, you may want your pipeline to output its result data to an external storage system. Beam provides read and write transforms for a [number of common data storage types](https://beam.apache.org/documentation/io/built-in/).

**v) Run the pipeline**

Run the pipeline using the designated Pipeline Runner. When you run your Beam driver program, the Pipeline Runner that you designate constructs a workflow graph of your pipeline based on the PCollection objects you’ve created and transforms that you’ve applied. That graph is then executed using the appropriate distributed processing back-end, becoming an asynchronous “job” (or equivalent) on that back-end.

**Different types of runners**

1. **Direct runner**

The Direct Runner executes pipelines on your machine and is designed to validate that pipelines adhere to the Apache Beam model as closely as possible. Instead of focusing on efficient pipeline execution, the Direct Runner performs additional checks to ensure that users do not rely on semantics that are not guaranteed by the model.

Using the Direct Runner for testing and development helps ensure that pipelines are robust across different Beam runners. In addition, debugging failed runs can be a non-trivial task when a pipeline executes on a remote cluster. Instead, it is often faster and simpler to perform local unit testing on your pipeline code. Unit testing your pipeline locally also allows you to use your preferred local debugging tools.

When executing your pipeline from the command-line, set runner to direct or DirectRunner. The default values for the other pipeline options are generally sufficient.Local execution is limited by the memory available in your local environment. It is highly recommended that you run your pipeline with data sets small enough to fit in local memory. You can create a small in-memory data set using a [Create](https://github.com/apache/beam/blob/master/sdks/python/apache_beam/transforms/core.py) transform, or you can use a [Read](https://github.com/apache/beam/blob/master/sdks/python/apache_beam/io/iobase.py) transform to work with small local or remote files.If your pipeline uses an unbounded data source or sink, you must set the streaming option to true.

1. **Apache Apex**

It executes Apache Beam pipelines using [Apache Apex](http://apex.apache.org/) as an underlying engine. The runner has broad support for the [Beam model and supports streaming and batch pipelines](https://beam.apache.org/documentation/runners/capability-matrix/).

[Apache Apex](http://apex.apache.org/) is a stream processing platform and framework for low-latency, high-throughput and fault-tolerant analytics applications on Apache Hadoop. Apex has a unified streaming architecture and can be used for real-time and batch processing.

1. **Apache Flink**

The Apache Flink Runner can be used to execute Beam pipelines using [Apache Flink](https://flink.apache.org/). For execution you can choose between a cluster execution mode or a local embedded execution mode which is useful for testing pipelines.

The Flink Runner and Flink are suitable for large scale, continuous jobs, and provide:

i) A streaming-first runtime that supports both batch processing and data streaming programs

ii) A runtime that supports very high throughput and low event latency at the same time

iii) Fault-tolerance with exactly-once processing guarantees

iv) Custom memory management for efficient and robust switching between in-memory and out-of-core data processing algorithms

It is important to understand that the Flink Runner comes in two flavors:

i) A legacy Runner which supports only Java (and other JVM-based languages)

ii) A portable Runner which supports Java/Python/Go

1. **Apache Samza**

The Apache Samza Runner can be used to execute Beam pipelines using [Apache Samza](http://samza.apache.org/). The Samza Runner executes Beam pipeline in a Samza application and can run locally. The application can further be built into a .tgz file, and deployed to a YARN cluster or Samza standalone cluster with Zookeeper.

The Samza Runner and Samza are suitable for large scale, stateful streaming jobs, and provide:

i) First class support for local state (with RocksDB store). This allows fast state access for high frequency streaming jobs.

ii) Fault-tolerance with support for incremental checkpointing of state instead of full snapshots. This enables Samza to scale to applications with very large state.

iii) A fully asynchronous processing engine that makes remote calls efficient.

iv) Flexible deployment model for running the the applications in any hosting environment with Zookeeper.

1. **Apache Spark**

The Apache Spark Runner can be used to execute Beam pipelines using [Apache Spark](http://spark.apache.org/). The Spark Runner can execute Spark pipelines just like a native Spark application; deploying a self-contained application for local mode, running on Spark’s Standalone RM, or using YARN or Mesos.

The Spark Runner executes Beam pipelines on top of Apache Spark, providing:

i) Batch and streaming (and combined) pipelines.

ii) The same fault-tolerance [guarantees](http://spark.apache.org/docs/latest/streaming-programming-guide.html#fault-tolerance-semantics) as provided by RDDs and DStreams.

iii) The same [security](http://spark.apache.org/docs/latest/security.html) features Spark provides.

iv) Built-in metrics reporting using Spark’s metrics system, which reports Beam Aggregators as well.

v) Native support for Beam side-inputs via spark’s Broadcast variables.

1. **Google Cloud DataFlow**

The Google Cloud Dataflow Runner uses the [Cloud Dataflow managed service](https://cloud.google.com/dataflow/service/dataflow-service-desc). When you run your pipeline with the Cloud Dataflow service, the runner uploads your executable code and dependencies to a Google Cloud Storage bucket and creates a Cloud Dataflow job, which executes your pipeline on managed resources in Google Cloud Platform.

The Cloud Dataflow Runner and service are suitable for large scale, continuous jobs, and provide:

i) a fully managed service

ii) [autoscaling](https://cloud.google.com/dataflow/service/dataflow-service-desc#autoscaling) of the number of workers throughout the lifetime of the job

iii) [dynamic work rebalancing](https://cloud.google.com/blog/big-data/2016/05/no-shard-left-behind-dynamic-work-rebalancing-in-google-cloud-dataflow)

**4.1.4 Windowing**

Windowing subdivides a PCollection according to the timestamps of its individual elements. Transforms that aggregate multiple elements, such as GroupByKey and Combine, work implicitly on a per-window basis — they process each PCollection as a succession of multiple, finite windows, though the entire collection itself may be of unbounded size.

In the Beam model, any PCollection can be subdivided into logical windows. Each element in a PCollection is assigned to one or more windows according to the PCollection’s windowing function, and each individual window contains a finite number of elements. Grouping transforms then consider each PCollection’s elements on a per-window basis. GroupByKey, for example, implicitly groups the elements of a PCollection by key and window.

**Windowing with bounded PCollections**

Windowing considers only the implicit timestamps attached to each element of a PCollection, and data sources that create fixed data sets (such as TextIO) assign the same timestamp to every element. This means that all the elements are by default part of a single, global window.We can assign your own timestamps to each element. To assign timestamps to elements, use a ParDo transform with a DoFn that outputs each element with a new timestamp.

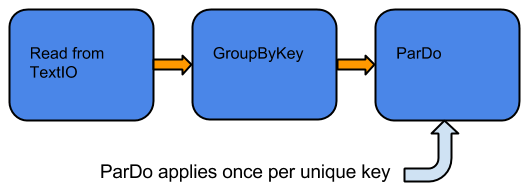


Fig. 4.3 GroupByKey and ParDo Without Windowing, on a Bounded Collection

Here the GroupByKey creates a collection of unique keys, and then ParDo gets applied exactly once per key. Even if you don’t set a windowing function, there is still a window – all elements in your PCollection are assigned to a single global window.

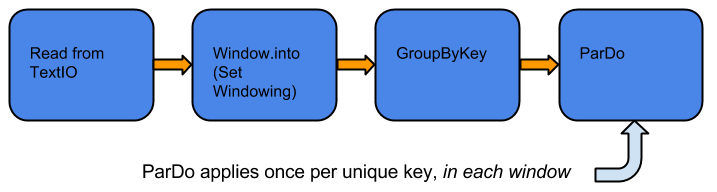


Fig 4.4 GroupByKey and ParDo With Windowing, on a Bounded Collection.

We set a [windowing function](https://beam.apache.org/documentation/programming-guide/#setting-your-pcollections-windowing-function) for the PCollection. The GroupByKey transform groups the elements of the PCollectionby both key and window, based on the windowing function. The subsequent ParDo transform gets applied multiple times per key, once for each window.

**Types of Windowing functions**

You can define different kinds of windows to divide the elements of your PCollection. Beam provides several windowing functions, including:

a) Fixed Time Windows

b) Sliding Time Windows

c) Per-Session Windows

d) Single Global Window

e) Calendar-based Windows (not supported by the Beam SDK for Python)

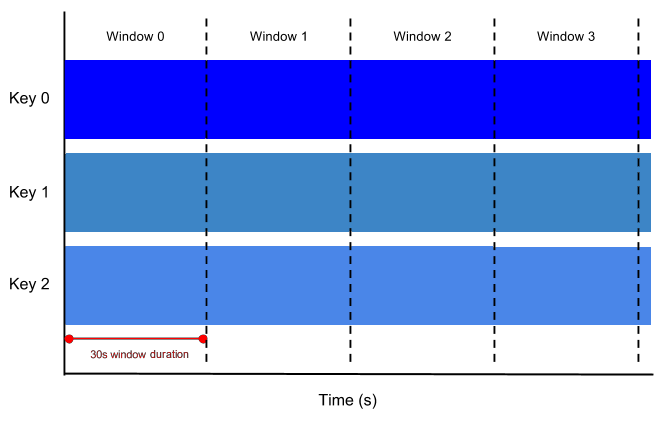
You can also define your own WindowFn if you have a more complex need.

Each element can logically belong to more than one window, depending on the windowing function you use.

1. **Fixed Time Windows**

A fixed time window represents a consistent duration, non overlapping time interval in the data stream.

Consider windows with a thirty second duration-All of the elements in your unbounded PCollection with timestamp values from 0:00:00 up to (but not including) 0:00:30 belong to the first window, elements with timestamp values from 0:00:30 up to (but not including) 0:01:00 belong to the second window, and so on.

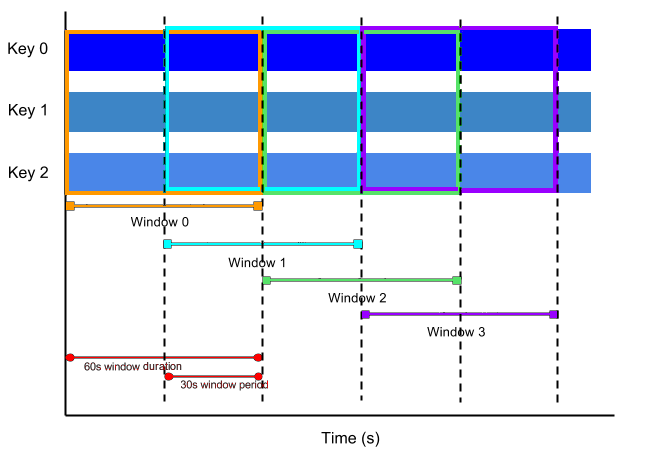


**Fig. 4.5** Fixed Time Windows, 30s in Duration

**b) Sliding Time Windows**

A sliding time window also represents time intervals in the data stream; however, sliding time windows can overlap. For example, each window might capture one minute worth of data, but a new window starts every thirty seconds. The frequency with which sliding windows begin is called the period. Therefore, our example would have window duration of one minute and a period of thirty seconds.

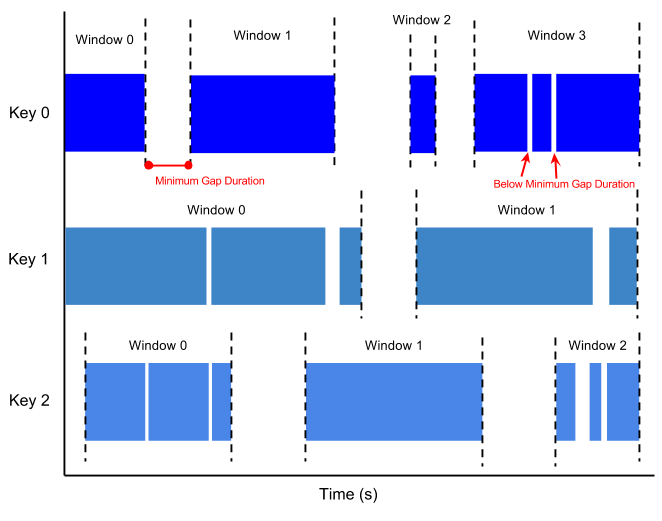
Because multiple windows overlap, most elements in a data set will belong to more than one window. This kind of windowing is useful for taking running averages of data; using sliding time windows, you can compute a running average of the past one minutes’ worth of data, updated every thirty seconds, in our example.



**Fig. 4.6** Sliding Time Windows, with 1 Minute Window Duration and 30s Window Period

**c) Session Windows**

A session window function defines windows that contain elements that are within a certain gap duration of another element. Session windowing applies on a per-key basis and is useful for data that is irregularly distributed with respect to time. For example, a data stream representing user mouse activity may have long periods of idle time interspersed with high concentrations of clicks. If data arrives after the minimum specified gap duration time, this initiates the start of a new window.

Fig. 4.7 Session Windows, with a Minimum Gap Duration.

**d) The single global window**

By default, all data in a PCollection is assigned to the single global window, and late data is discarded. If your data set is of a fixed size, you can use the global window default for your PCollection.

You can use the single global window if you are working with an unbounded data set (e.g. from a streaming data source) but use caution when applying aggregating transforms such as GroupByKey and Combine. The single global window with a default trigger generally requires the entire data set to be available before processing, which is not possible with continuously updating data. To perform aggregations on an unbounded PCollection that uses global windowing, you should specify a non-default trigger for that PCollection.

**4.1.5 Watermarks in Apache Beam**

In any data processing system, there is a certain amount of lag between the time a data event and the time the actual data element gets processed at any stage in your pipeline. In addition, there are no guarantees that data events will appear in your pipeline in the same order that they were generated.

For example, let’s say we have a PCollection that’s using fixed-time windowing, with windows that are two minutes long. For each window, Beam must collect all the data with an event time timestamp in the given window range (between 0:00 and 1:59 in the first window, for instance). Data with timestamps outside that range (data from 2:00 or later) belong to a different window.

However, data isn’t always guaranteed to arrive in a pipeline in time order, or to always arrive at predictable intervals. Beam tracks a watermark, which is the system’s notion of when all data in a certain window can be expected to have arrived in the pipeline. Once the watermark progresses past the end of a window, any further element that arrives with a timestamp in that window is considered late data.

For example, suppose we have a simple watermark that assumes approximately 30s of lag time between the data timestamps and the time the data appears in the pipeline, then Beam would close the first window at 2:30. If a data record arrives at 2:35, but with a timestamp that would put it in the 0:00-1:59 window (say, 1:38), then that record is late data.

**Managing late data**

You can allow late data by invoking the .withAllowedLateness operation when you set your PCollection’s windowing strategy. The following code example demonstrates a windowing strategy that will allow late data up to two days after the end of a window.

When you set .withAllowedLateness on a PCollection, that allowed lateness propagates forward to any subsequent PCollection derived from the first PCollection you applied allowed lateness to. If you want to change the allowed lateness later in your pipeline, you must do so explictly by applyingWindow.configure().withAllowedLateness().

Managing late data is not supported in the Beam SDK for Python.

**4.1.6 Triggers**

When collecting and grouping data into windows, Beam uses triggers to determine when to emit the aggregated results of each window.

You can set triggers for your PCollections to change the default behavior. Beam provides a number of pre-built triggers that you can set:

**a) Event time triggers:** These triggers operate on the event time, as indicated by the timestamp on each data element. Beam’s default trigger is event time-based.

**b) Processing time triggers:** These triggers operate on the processing time – the time when the data element is processed at any given stage in the pipeline.

**c) Data-driven triggers:** These triggers operate by examining the data as it arrives in each window, and firing when that data meets a certain property. Currently, data-driven triggers only support firing after a certain number of data elements.

**d) Composite triggers:** These triggers combine multiple triggers in various ways.

At a high level, triggers provide two additional capabilities compared to simply outputting at the end of a window:

a) Triggers allow Beam to emit early results, before all the data in a given window has arrived. For example, emitting after a certain amount of time elapses, or after a certain number of elements arrives.

b) Triggers allow processing of late data by triggering after the event time watermark passes the end of the window.

These capabilities allow you to control the flow of your data and balance between different factors depending on your use case:

1. Completeness:
2. Latency
3. Cost

For example, a system that requires time-sensitive updates might use a strict time-based trigger that emits a window every N seconds, valuing promptness over data completeness. A system that values data completeness more than the exact timing of results might choose to use Beam’s default trigger, which fires at the end of the window.

**4.2 Model of Proposed System**

Stream Data to PubSub

Fetching Data from Google Finance Using Excel Query

Excel Data

VBA Script to Refresh and Save Data at Every Regular Time Interval

Read Data from ExcelSheet and Publish to PubSub using Python Script

DataFlow Runner

Read Data from PubSub

Prediction Transform

Extract Parameters for Prediction from Input

Prediction using Regression Model

Formatting According to Required Format

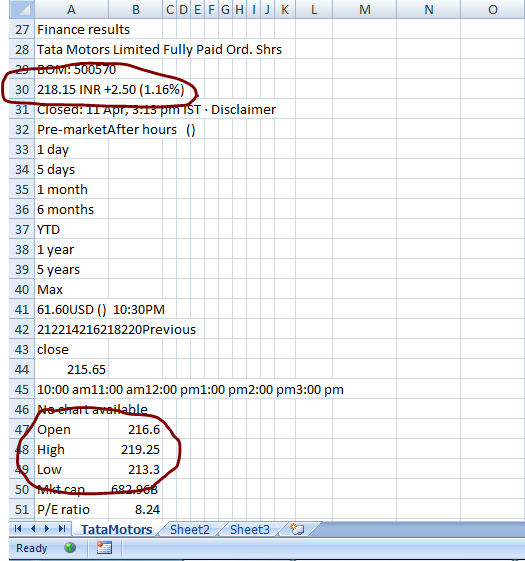
Write Output To PubSub

Output Data To Cloud Storage from PubSub Using DataFlow Job

**Chapter 5 Implementation Detail**

**5.1 Streaming Input to PubSub**

1. Fetch data in ExcelSheet from Google finance using Excel Query.
2. Continuously update Sheet using VBA Scripting
   1. Refresh connection to get fresh data in sheet.
   2. Save sheet to use fresh data using python.
   3. Repeat above steps again after 30 seconds.



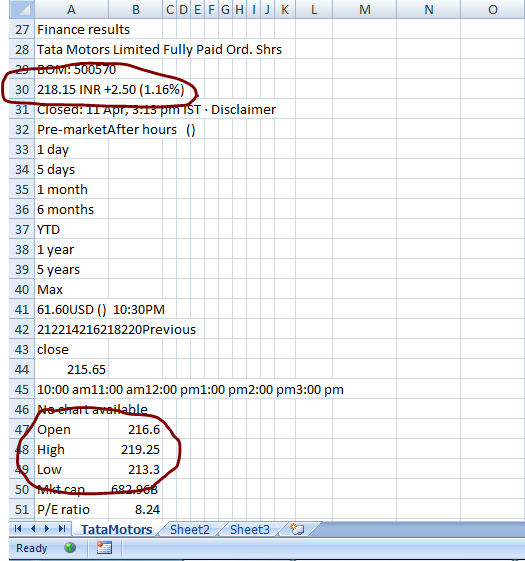
****

Fig. 5.1 Fetched Data in Excel

1. Python script to stream data to PubSub
   1. Read data from ExcelSheet.
   2. Format the parameters.
   3. Publish to PubSub.

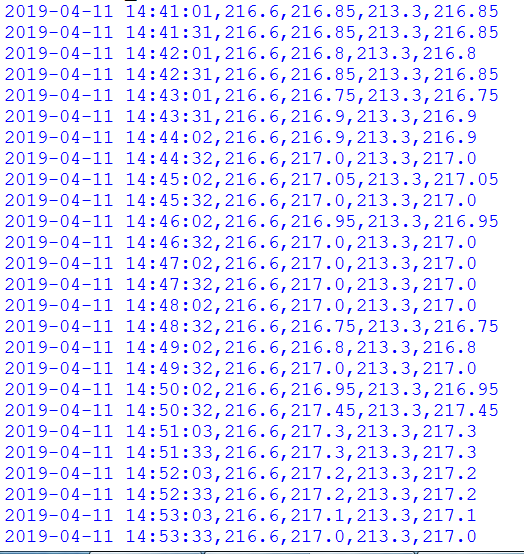


Fig 5.2 Streaming Data to PubSub

**5.2 DataFlow Runner**

1. Import libraries related to
   1. Apache Beam Pipeline, Windowing and Triggering
   2. Python Machine Learning and Data Manipulation Libraries
2. Create the Regression Model and train it using historical data.
3. Create the Apache Beam Pipeline and set the pipeline Options.
4. Read the streaming input from PubSub using SDK function “beam.io.ReadFromPubSub()” and store into PCollection object.
5. Create a ParDo Transform Prediction
6. Split the PCollection object received from the pipeline using comma delimiters.
7. Send the Parameters to the Regression Model for Prediction.
8. Return the predicted value in the form of PCollection object back to the pipeline.
9. Create a Format Transform to format the predicted output received into the format required for publishing the data to PubSub.
10. Publish the output data to PubSub using SDK function “beam.io.WriteStringsToPubSub()”.

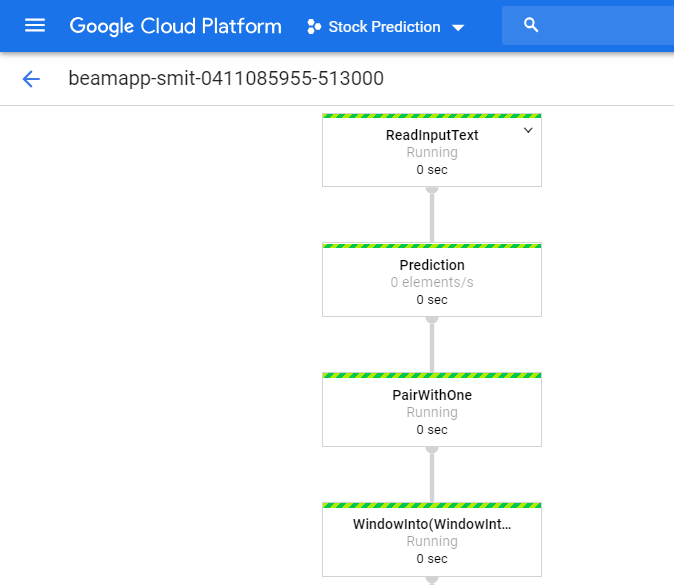
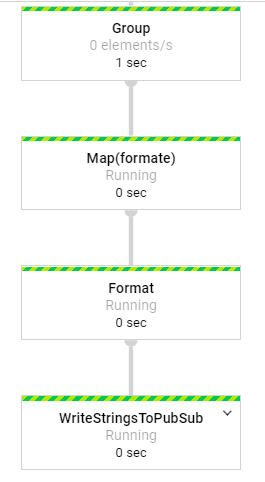
 

Fig. 5.3 DataFlow job Running on DataFlow Runner on Google Cloud.

**5.3 Write to Google Cloud Storage from PubSub**

1. Use Google Cloud Console to create DataFlow job.
2. Create a DataFlow job that fetches data from PubSub and outputs it to the Google Cloud Storage every five minutes.
3. Terminate all the DataFlow jobs once the results are acquired.

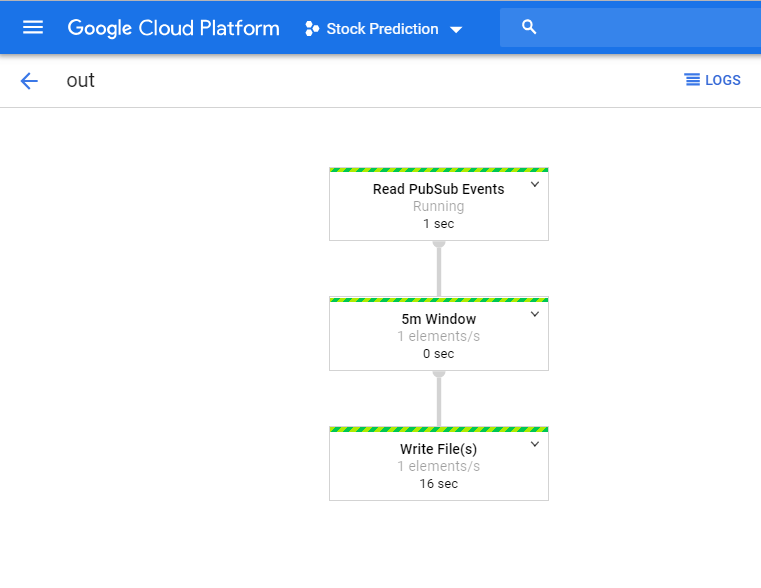
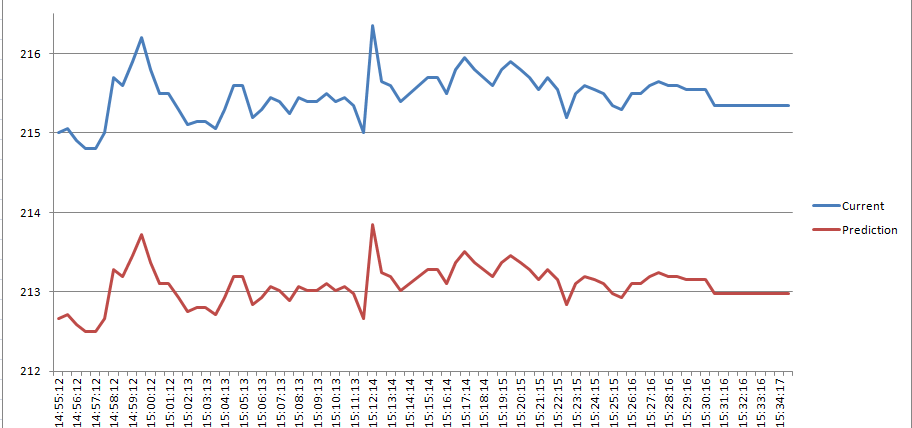


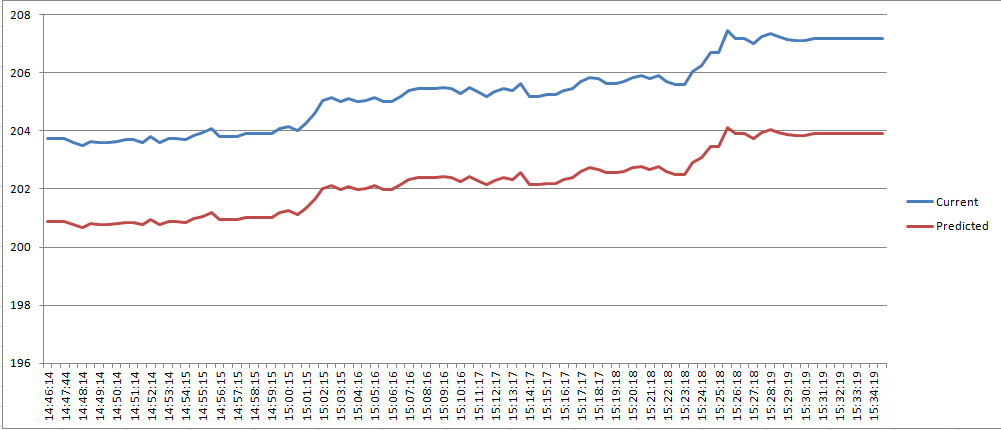
Fig. 5.4 DataFlow job that Outputs Data from PubSub To Google Cloud Storage.

**Chapter 6 Experimental Results**

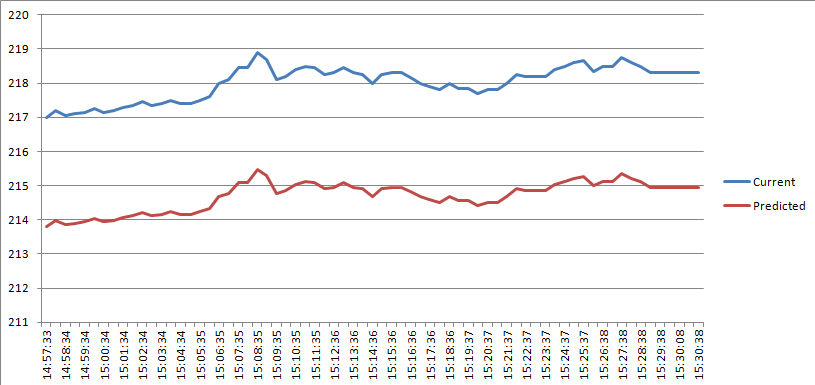
**09/04**

****

**10/04**

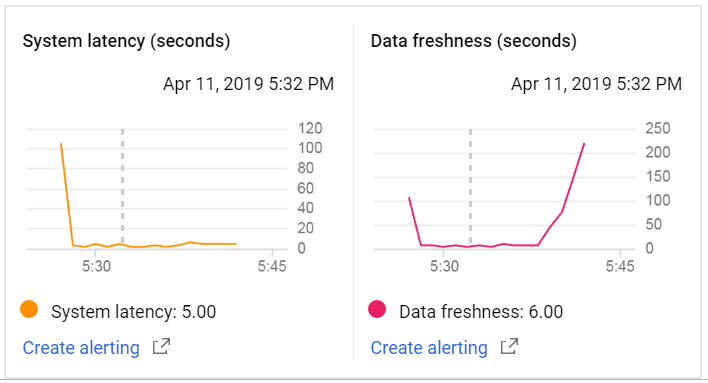
****

**11/04**

****

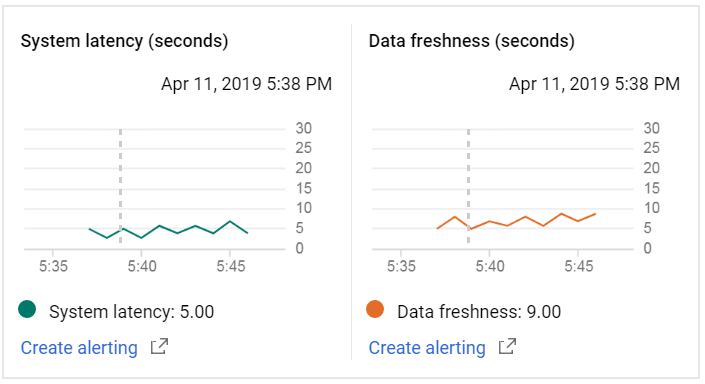
**15 seconds**

|  |  |  |
| --- | --- | --- |
| After minutes | System Latency(Seconds) | Data Freshness(Seconds) |
| 1 | 106 | 107 |
| 2 | 4 | 9 |
| 3 | 3 | 8 |
| 4 | 5 | 6 |
| 5 | 3 | 8 |
| 6 | 5 | 6 |
| 7 | 3 | 8 |
| 8 | 2 | 6 |
| 9 | 4 | 10 |
| 10 | 3 | 7 |

****

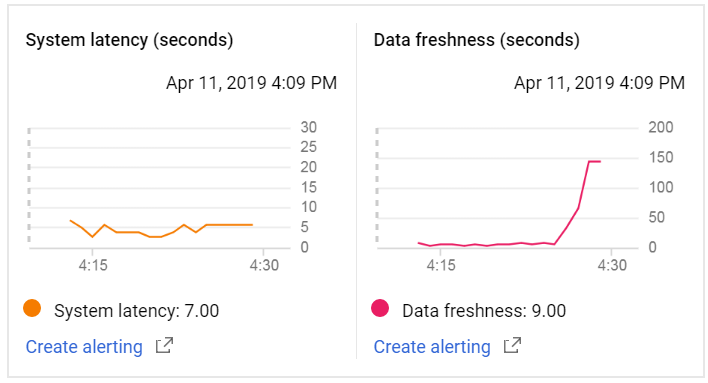
**Trigger**

|  |  |  |
| --- | --- | --- |
| After minutes | System Latency(Seconds) | Data Freshness(Seconds) |
| 1 | 5 | 5 |
| 2 | 3 | 8 |
| 3 | 5 | 5 |
| 4 | 3 | 7 |
| 5 | 6 | 6 |
| 6 | 4 | 8 |
| 7 | 6 | 6 |
| 8 | 4 | 9 |
| 9 | 7 | 7 |
| 10 | 4 | 9 |

****

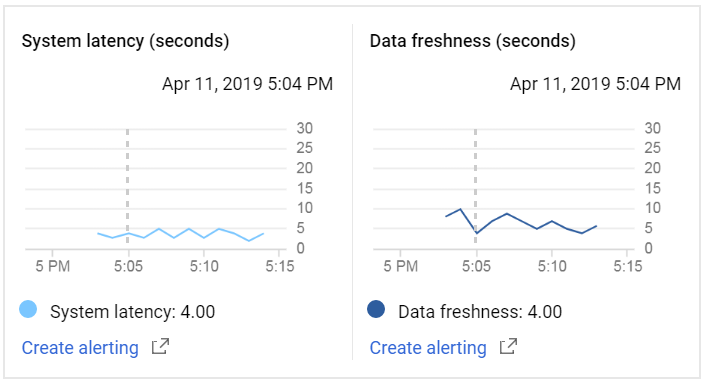
**30 seconds**

|  |  |  |
| --- | --- | --- |
| After minutes | System Latency(Seconds) | Data Freshness(Seconds) |
| 1 | 7 | 9 |
| 2 | 7 | 9 |
| 3 | 5 | 5 |
| 4 | 3 | 7 |
| 5 | 6 | 6 |
| 6 | 4 | 4 |
| 7 | 4 | 6 |
| 8 | 4 | 4 |
| 9 | 3 | 7 |
| 10 | 3 | 6 |

****

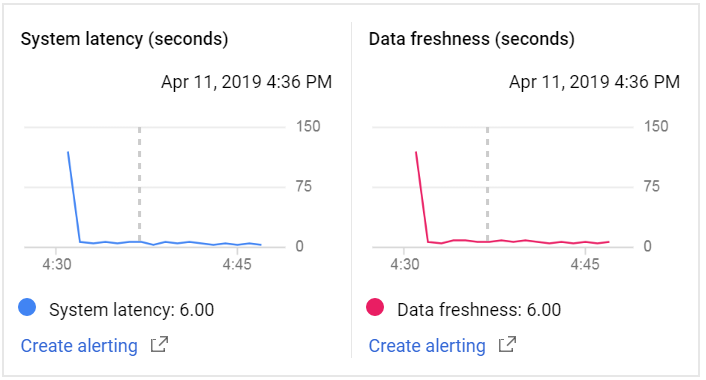
**45 Seconds**

|  |  |  |
| --- | --- | --- |
| After minutes | System Latency | Data Freshness |
| 1 | 4 | 8 |
| 2 | 3 | 10 |
| 3 | 3 | 7 |
| 4 | 5 | 9 |
| 5 | 3 | 7 |
| 6 | 5 | 5 |
| 7 | 3 | 7 |
| 8 | 5 | 5 |
| 9 | 4 | 4 |
| 10 | 2 | 6 |

****

**60 Seconds**

|  |  |  |
| --- | --- | --- |
| After minutes | System Latency | Data Freshness |
| 1 | 120 | 120 |
| 2 | 7 | 7 |
| 3 | 6 | 8 |
| 4 | 4 | 9 |
| 5 | 7 | 7 |
| 6 | 6 | 6 |
| 7 | 3 | 8 |
| 8 | 6 | 6 |
| 9 | 6 | 6 |
| 10 | 5 | 5 |

****