## Batch Processing

**Batch processing** is where the processing happens of blocks of data that have already been stored over a period of time. For example, processing all the transactions that have been performed by a major financial firm in a week. This data contains millions of records for a day that can be stored as a file or record etc. This particular file will undergo processing at the end of the day for various analysis that the firm wants to do. Obviously it will take a large amount of time for that file to be processed. That would be what Batch Processing is :)

Hadoop MapReduce is the best framework for processing data in batches. The following figure

gives you detailed explanation how Hadoop processing data using MapReduce.

Batch processing works well in situations where you don’t need real-time analytics results, and

when it is more important to process large volumes of data to get more detailed insights than it is to

get fast analytics results.

# What is Stream Processing

# Stream processing is a golden key if you want analytics results in real time. Stream processing allows us to process data in real time as they arrive and quickly detect conditions within small time period from the point of receiving the data. Stream processing allows you to feed data into analytics tools as soon as they get generated and get instant analytics results. There are multiple open source stream processing platforms such as Apache Kafka, Apache Flink, Apache Storm, Apache Samza, etc. I would recommend [**WSO2 Stream Processor (WSO2 SP)**](https://wso2.com/analytics)**,** the open source stream processing platform which I have helped built. WSO2 SP can ingest data from Kafka, HTTP requests, message brokers. You can query data stream using a [*“Streaming SQL”*](https://wso2.com/library/articles/2018/02/stream-processing-101-from-sql-to-streaming-sql-in-ten-minutes/) language. With just two commodity servers it can provide high availability and can handle 100K+ TPS [*throughput*](https://medium.com/@gowthamy/wso2-stream-processor-performance-analysis-internship-project-01-c29a93d58865). It can scale up to millions of TPS on top of Kafka. Furthermore, the Business Rules Manager of WSO2 SP allows you to define templates and generate business rules from them for different scenarios with common requirements.

fraud detection. If you stream-process transaction data, you can detect anomalies that signal fraud in real time, then stop fraudulent transactions before they are completed.

In Batch Processing it processes over all or most of the data but In Stream Processing it processes over data on rolling window or most recent record. So Batch Processing handles a large batch of data while Stream processing handles Individual records or micro batches of few records.

In the point of performance the latency of batch processing will be in a minutes to hours while the latency of stream processing will be in seconds or milliseconds.

At the end of the day, a solid developer will want to understand both work flows. It’s all going to come down to the use case and how either work flow will help meet the business objective.

·

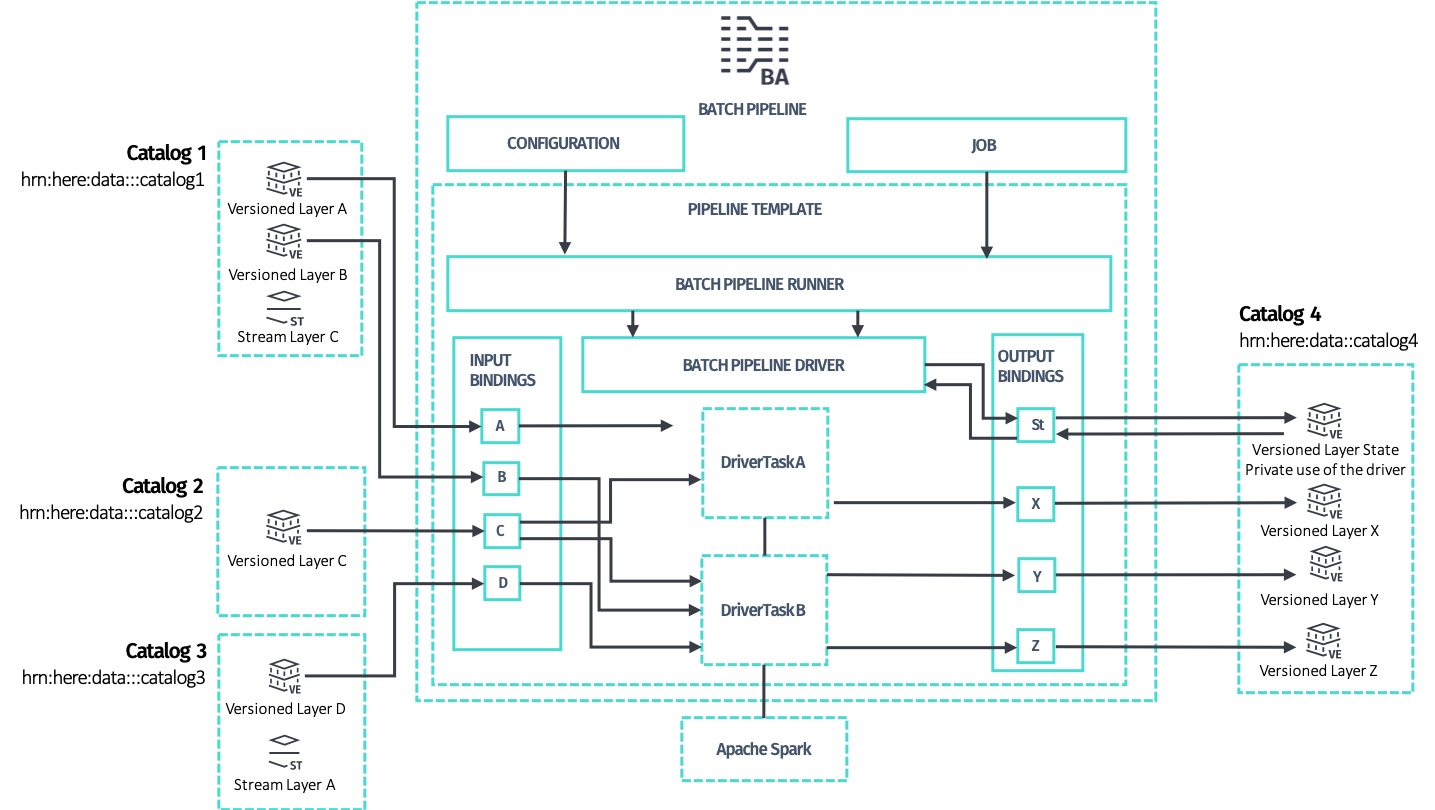
Batch pipelines are a particular type of pipelines used to process data in batches.

When a pipeline is in the SCHEDULED state, the Pipeline Service triggers its execution when some conditions are met or external events are detected, such as a change in the input catalogs. A *job description* holds the details of this change and is passed on to the pipeline. Once triggered, the pipeline switches to the RUNNING state, processes the data and commits the result to the output catalog. This process terminates and the pipeline returns to the SCHEDULED state.

For more information on pipelines, their state, and state transitions, see the *Pipeline Service Developer's Guide*.

The Data Processing Library supports the development of batch pipelines on Apache Spark.

Internal Structure



To develop a batch pipeline, start with these two classes:

1. PipelineRunner, the runnable class that implements the main method and provides parsing of the command line and system properties to interface with Pipeline Service. Pipeline configuration and job description are detected and provided to developer's code. Developers should use this class as entry point both for running the code locally and for deploying the pipeline to the Pipeline Service.

2. Driver, the main class that controls the distributed processing, provides access to Apache Spark and preconfigured Spark-friendly access to the Data API of input and output catalogs in what is called

## Configure the Driver

The Driver can be configured either manually or via the DriverBuilder to have one or more data processing DriverTasks attached to it. Each task represents an end-to-end batch processing logic that produces data for a set of output layers by consuming data from a set of input layers.

Each task specifies:

· a subset of layers for the input catalogs which the task intends to process -- a task declares as input one or more layers among the layers available in the input catalogs. Each layer is specified by the catalog's symbolic ID and the layer ID. Multiple tasks may declare the same input layer as input, since it is safe for a set of input layers of multiple tasks to overlap.

· which output layers in the output catalog for the task to generate: each task is responsible for producing one or more output layers in an exclusive way, meaning that each output layer can be produced by one task only. Two or more tasks *cannot* declare the same output layer, as this is configuration is not supported.

The Data Processing Library supports incremental processing, such that a task does not run as long as no changes are detected in its input layers. However, other tasks may still run.

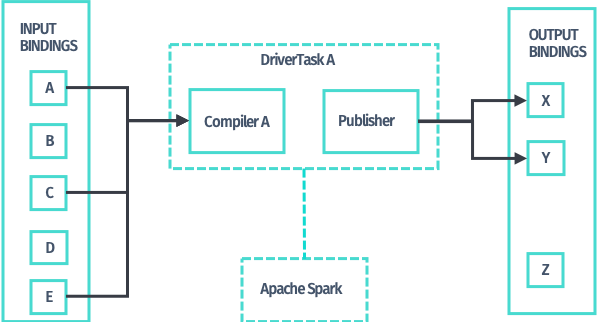
## **Compilation as a Form of Data Processing**

While you can directly implement DriverTask, we recommend implementing *compilers* instead, as this is a higher-level of abstraction provided by the processing library.

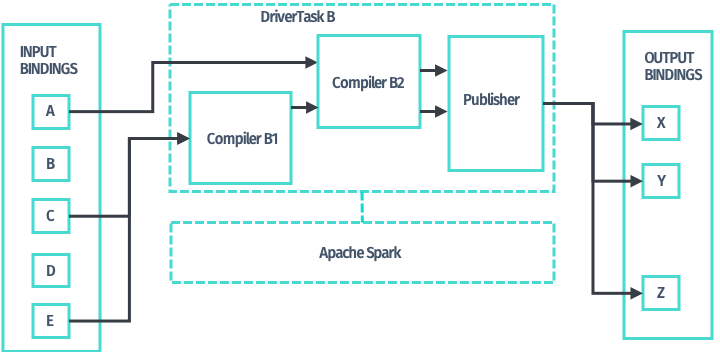
In the context of the Data Processing Library, a *compiler* refers to batch pipelines that functionally transform input layers to output layers. This type of pipelines can only operate with input and output versioned layers.

This type of transformation is referred to as *compilation* or more specifically, *map compilation*, where the layers involved represent input and output map data in standard or custom formats.

The Data Processing Library not only helps you write *compilers*, but also guides you with [patterns](https://developer.here.com/olp/documentation/data-processing-library/dev_guide/content/topics/patterns.html) that enable important features, such as [incremental compilation](https://developer.here.com/olp/documentation/data-processing-library/dev_guide/content/topics/incremental-compilation.html).



When you have expressed the required data processing logic in the form of a compiler, the processing library provides implementations of DriverTask to apply that data transformation logic to the input layers declared for a task; you do not have to write any additional code. Use the newTaskBuilder method of DriverBuilder and the TaskBuilder it returns.



In addition, ***compilers*** may be [*chained together*](https://developer.here.com/olp/documentation/data-processing-library/dev_guide/content/topics/chaining-patterns-together.html), so that you can use the output layers produced by one ***compiler*** as input of any other ***compiler*** down the chain. Use newMultiCompilerTaskBuilder method of DriverBuilder and the corresponding MultiCompilerTaskBuilder it returns to obtain a DriverTask implementation to chain ***compilers***.

## **Distributed Processing**

Once you have configured the Driver by assigning one or more DriverTask objects to it, the PipelineRunner starts processing. The Driver begins a new publication on the output catalog via its Data API and then launches each task on Spark, sequentially, usually one Spark job per task. This may be different for your custom DriverTasks and for [RDD-based](https://developer.here.com/olp/documentation/data-processing-library/dev_guide/content/topics/rdd-based-patterns.html) compilation patterns.

The Driver only handles version information and orchestrates the processing. Neither metadata nor data payloads is processed by the Driver, as doing so would hinder the scalability of the solution.

Sequentially, for each task:

1. The Driver queries the Data API for the layers' metadata which the task requires, optionally filters it, and passes it to the actual DriverTask implementation. This process consists of Spark transformations that are run in a distributed manner. Typically, the task implementation uses the Retrievers located in a DriverContext to retrieve the payloads from the Data Blob API. Retrieving also runs in parallel, as part of the Spark job. The task continues processing the data by decoding the Protobuf payloads and applying custom processing logic to it.

2. The compilation task produces payloads that should be committed to the layers of the output catalog which the task owns. The result of processing is returned by the *compiler* implementation and is handed over to a built-in incremental publisher, always as part of the same Spark job.

3. The [incremental publisher](https://developer.here.com/olp/documentation/data-processing-library/dev_guide/content/topics/payloads.html#publishing-payloads) has access to the output catalog metadata to compare the produced payloads with what is already available via checksums. The incremental publisher then uploads payloads that are different to the Data Blob API; payloads that are the same are discarded. This process is always implemented in a distributed way, as part of the Spark job. The incremental publisher also produces the commit metadata to be used with the Data Publish API.

When the Driver has successfully finished running all of its attached DriverTasks, the publication is now complete and the new data plus metadata are published transactionally to the output catalog. If an error occurs, the publication is aborted and the job is marked as FAILED.