Technical University of Crete
Electrical and Computer Engineering

Cross-Platform Query Optimization on Apache Beam

by Giorgos Stamatakis

Thesis Committee:
Prof. Antonios Deligiannakis (Supervisor)
Prof. Minos Garofalakis
Prof. Vasilis Samoladas

Abstract

We live in an era where the rapid growth of data streams both in complexity and velocity introduce new challenges on a daily basis. These streams can be infinite, emit data at high-speeds and can be generated by non-stationary distributions, thus a modern approach is required when performing complex queries over such streams. Since modern Big Data stream processing frameworks are evolving rapidly, high-level languages and abstractions are necessary in order to provide support for increasingly complex queries. Query processing however, requires a high-level language be translated into a set of low-level data manipulation operations and on top of that producing an optimal plan for the given query can be extremely difficult as it requires minimizing costs based on available statistics. The objective of query optimization is to maximize (or minimize) metrics like throughput and latency which are vital to the performance of a stream processing system but in order to do so effectively a framework needs to track a wide variety of performance metrics of components inside and outside its cluster. Metrics can include the health and utilization of the stream pipeline operators, network speed and performance, hardware statistics of each cluster node and finally information about incoming data streams like skew, throughput and tuple size.

In this thesis we propose a high-level real-time query optimization platform that collects various statistics from modern stream processing frameworks and based on the requested queries each one is assigned on the most suitable framework. Such strategies aim to combine the metrics and statistics collected by our platform, both past and present, in order to correctly match queries to available frameworks. Our proposed platform consists of three main modules, the first of which is responsible for gathering available metrics from resource managers and available frameworks. The second module consists of a data ingestion Kafka Streams application that allow us to perform distributed sampling over incoming data streams and provide us with information related to incoming data streams, like skew and throughput. Finally, based on the available metrics each query is assigned to a framework running a pipeline designed on Apache Beam, a framework allowing us to write high-level code that can be executed in a variety of Big Data frameworks with ease.

Experimental results prove that the gathered statistics improve query assignments as well as overall performance metrics like throughput and latency. Furthermore, increasing the parallelism level of frameworks yields better results as higher rates of data can be ingested by the stream processing frameworks. Finally, experimental evaluation shows that different strategies may work better under certain conditions.

Acknowledgements

I would like to thank my supervisor Prof. Antonios Deligiannakis for the help and feedback he eagerly provided as well as the rest of the thesis committee Prof. Vasilis Samoladas and Prof. Minos Garofalakis for their support. I would also like to thank my friends and family for their encouragement and support over the years.

Table of contents

Abstract ii

Acknowledgements iii

List of figures vi

1. Introduction 1

 1.1 Thesis Outline 2

2. Apache Kafka 3

 2.1 Overview 3

 2.2 Records 3

 2.3 Topics 3

 2.4 Partitions 4

 2.5 Replication and fault-tolerance 4

 2.6 Producers 4

 2.7 Consumers 5

 2.8 Kafka Streams 5

  2.8.1 Architecture 5

  2.8.2 State stores 6

3. Stream processing 7

 3.1 Apache Beam 8

  3.1.1 Overview 8

  3.1.2 Parallel Collections 9

  3.1.3 Transformations 9

 3.1.4 Beam I/O 12

 3.1.5 Windowing 12

 3.1.6 Watermarks and Triggers 15

4. Platform implementation and metrics collection 17

 4.1 Kafka Streams topology 17

 4.1.1 Reservoir sampling 18

 4.2 Statistics and query assignment 20

 4.2.1 Metrics and statistics acquisition 20

 4.2.2 Query assignment 23

 4.3 Apache Beam pipeline and Runners 24

5. Experimental Evaluation 28

5.1 Cluster Setup....28
5.2 Scalability....31
5.3 Strategy comparison....35
6. Conclusions and Future Work....39

List of figures

Figure 1: A Kafka Topic with 3 partitions with new records being added. 4
Figure 2: Two Kafka producers with two uneven consumer groups. 5
Figure 3: A stream with two topics and two stream tasks. 6
Figure 4: Two stream tasks with their dedicated local state stores. 6
Figure 5: Event, Ingestion and Processing times. 7
Figure 6: Transform chaining of an input parallel collection. 10
Figure 7: Windowing and grouping in succession. 13
Figure 8: Example of fixed time windows on three different streams. 13
Figure 9: Fixed time windows. 14
Figure 10: Example of a session time window. 14
Figure 11: Kafka topics and the proposed Kafka Streams topology. 18
Figure 12: Algorithm R pseudocode. 18
Figure 13: Distributed Algorithm R pseudocode. 19
Figure 14: Data skew of random variable x 21
Figure 15: Examples of skewed distributions. 22
Figure 16: Kurtosis of random variable x. 22
Figure 17: Different values of Kurtosis. 22
Figure 18: Overview of the Controller sub-module. 23
Figure 19: Projections of 2 close circles and 2 distant squares projected in this paper surface. 24
Figure 20: Extraction of query IDs. 25
Figure 21: CGBK and Join operations. 25
Figure 22: The processing latency of the Kafka Streams application for different number of parallel instances and input stream rates. 31
Figure 23: Spark runner throughput for varying parallelism degrees and stream rates. 32
Figure 24: Spark runner latency for different sets of parallelism degrees and stream rates. 32
Figure 25: Flink Runner throughput for varying parallelism degrees and stream rates. 33
Figure 26: Flink Runner latency for different sets of parallelism degrees and stream rates. 33
Figure 27: Apex Runner throughput for varying parallelism degrees and stream rates. 34
Figure 28: Apex Runner latency for different sets of parallelism degrees and stream rates. 34
Figure 29: Strategy latencies under input stream rate of 500 msg/s. 35
Figure 30: Strategy latencies under input stream rate of 2000 msg/s. 36
Figure 31: Strategy latencies under input stream rate of 5000 msg/s. 37
Figure 32: Average latency for each strategy under various stream rates. 38

1. Introduction

In the current day and age, the volume and velocity of generated information is increasing rapidly and on a daily basis while demands for accurate and real-time analytics are becoming more and more necessary. Sources that produce high speed data streams can be financial transactions, IoT sensors, click streams and various embedded systems all of which produce data that needs to be cleansed, stored and finally processed in order to extract valuable information. To perform a task like that a number of stream processing frameworks and middleware are used in order to provide guarantees over the speed and correctness of the various computations.

Stream processing frameworks are developed with having in mind unbounded data sources that emit data at varying rates and volumes. Processing of ingested data is done in a fully distributed environment with semantics that include various fault tolerant mechanisms and fallback scenarios in order to ensure that data is never lost. Due to the size and speed of such data streams as well as the distributed nature of the stream processing frameworks, queries usually consist of simple transformations that execute one-pass algorithms over incoming data before sending the results downstream. Environments like these usually express queries in the form of a directed acyclic graph (DAG) with stream operators as nodes and data streams as edges. The process of translating user code to a DAG can be a complex and lengthy process that involves multiple optimization steps and techniques as in order to maximize the efficiency of a stream processing pipeline an optimizer needs to take into account available statistics and metrics of both incoming and outgoing data. Finally, depending on the architecture of each framework each query can be translated into different low-level operations which best suit their available resources and statistics, which can result into the same query being executed faster and more efficiently in some frameworks under specific circumstances (e.g. skewed data).

A stream processing framework can gather statistics from incoming streams like throughput and latency as well as estimates about the distribution of data. Sampling mechanisms can also be employed in order produce summaries of incoming data that can provide information like skewness and cardinality. Furthermore, most stream processing frameworks and resource managers provide information for each of their nodes that can include hardware and network statistics which in turn helps users determine the quality and performance of their distributed algorithms. Depending on the framework a variety of statistics can be acquired for stream operators and their parallel instances that usually includes latency, throughput, rate of incoming and outgoing tuples and backpressure status, all of which can usually be provided for each operator instance or operator task. In conclusion, each stream processing framework handles queries differently depending on its architecture, the work load and available statistics which results in some queries performing significantly better on specific frameworks.

What we propose is a platform that assigns queries to a number of stream processing frameworks in order to increase performance based on available metrics and statistics. Since Apache Beam was used to generate high-level code that can be then ran on a variety of frameworks, also known as runners, for this thesis three stream processing frameworks were used in order to demonstrate our platform's capabilities: Apache Spark, Apache Flink and Apache Apex. The first part of our platform consists of an application that monitors every available framework and resource manager with the goal of collecting statistics and metrics from various public APIs, a process which will play a vital role later on with query assignments. The second part of the pipeline consists of a Kafka Streams distributed application with queryable state that acts as an ingestion mechanism which can also perform sampling on incoming data streams. More specifically this application monitors the size and speed of incoming data streams but also performs various normality tests over the ingested data stream all of which can be accessed via a REST API from other parts of the pipeline. When a new query arrives, our platform decides

what the most suitable framework will be, based on available statistics, and assigns the query to the respective framework. The third and final part of our platform consists of Apache Beam pipelines that run separately on all of our frameworks and each time a query is assigned to one of them data streams from the Kafka application are redirected to that pipeline. Furthermore, the act of assigning queries to frameworks consists of gathering some available at the time statistics into a multi-dimensional vector and then performing nearest neighbor queries in order to retrieve vectors of similar past queries that yielded the optimal results. Finally, when a number of similar past queries have been found one can chose to execute the query on the framework that yielded the best results according to their use case, i.e. the query that maximized (or minimized) their desired performance metrics.

1.1 Thesis Outline

In **chapter 2**, a description of the Apache Kafka middleware and the Kafka Streams application are given, briefly explaining their key features that will play a vital role in our proposed platform. Moreover, the distributed nature and stateful operations will also be discussed and compared with features that similar frameworks provide.

In **chapter 3**, the dataflow model is briefly explained along with some of its strengths and weaknesses. Furthermore, an introduction to Apache Beam will cover concepts that will be used when developing pipelines for other stream processing frameworks, also some core transformations and operators will be discussed.

During **chapter 4** we will expand on how our platform collects statistics and other vital information in order to efficiently assign queries to available frameworks. During this chapter we will also explain how individual subsystems are combined in order to create a platform for efficient cross platform query optimization. More specifically, we explain how the Kafka Streams application performs distributed sampling efficiently, what statistics are collected and how decision-making is affected. Furthermore, we explain how decision-making works and more specifically the clustering algorithm that is used to find past queries with similar statistics that performed well in order to assign each query to the best available framework. Finally, we discuss about how the queries are implemented on Apache Beam and how our platform leverages Kafka pub/sub distributed queues to deliver data-streams and queries to the optimal at the time framework.

Chapter 5 consists of experimental evaluations and results from various work-loads, frameworks and degrees of parallelism. Results show how throughput and latency change for varying degrees of parallelism and work-load. Lastly, we provide an example of Apache Beam direct-runner and how it enables the integration of more specialized hardware into stream processing pipelines with minimal effort.

Chapter 6 focuses on conclusions drawn from experimental results and possible future work.

2. Apache Kafka

2.1 Overview

Apache Kafka [1] is a distributed streaming platform originally open-sourced by LinkedIn and now one of the most popular Apache top level projects and is used in production by many companies due to its capabilities. Kafka enables applications to publish and subscribe to streams of records, similar to a message queue or enterprise messaging system while also providing fault-tolerant record stores in a durable way without sacrificing its scalability or latency. Kafka is mostly used by applications that transform fast data streams, require reliable message passing and benefit from having a flexible ingestion mechanism while maintaining the ability to scale on demand. Kafka has a unique architecture that borrows heavily from that of a distributed message queue and a real-time streaming platform which in turn results in a unique design with familiar terms. Furthermore, Kafka is meant to be ran on a fully distributed environment spanning multiple servers, also known as brokers, that form a cluster with a Zookeeper instance acting as a coordinator. Finally, such clusters contain records grouped into arbitrary categories called topics which form the basic unit of a simple but powerful Pub/Sub system.

2.2 Records

A record is simply an array of bytes that consists of a key-value pair, a timestamp and metadata that contain information about its properties (e.g. size, offset, host). All records are stored durably and can be read deterministically in case of a failure but in order to maximize availability and performance within a cluster records can also be replicated (mirrored) and distributed to more than one brokers, effectively removing a single point of failure. It’s worth mentioning that although the key-value pair of each message can be set arbitrarily, a JSON-like schema can also be enforced if consistency of record values is necessary. Note that the integrity of each record, along with other security checks, is automatically handled by the Zookeeper instance and the Kafka broker(s) as each record contains the necessary metadata required for such checks.

2.3 Topics

Topics are groups of records with a unique id and configuration settings created by the user. Topic structure is similar to that of a distributed queue and thus a single topic exists on one or more brokers. Incoming records are stored in order by the broker(s) which take advantage of the append-only property of a distributed message queue. More specifically producers write records to the queue tail while consumers can pull records from the queue at a different pace which lays the foundation of scalable high throughput of a distributed message passing system.

2.4 Partitions

Kafka topics can be divided into partitions by splitting their data to disjoint sets and distributing them to multiple brokers, a process that allows a topic to be parallelized seamlessly and on demand. Although each topic can have a near infinite number of consumers, multiple partitions allow consumers to read a topic in parallel as each partition is sent only to a subset of the original consumers, a process which drastically increases system throughput. However, consumers of a topic with multiple partitions result in messages being read out-of-order, a problem which can't be solved due the durable and distributed nature of this problem. Partitions are also Kafka's way of providing fault tolerance and scalability since they can be distributed among different brokers that may lie on different servers which provides redundancy and allows for horizontal scaling. Finally, each message per partition has an offset value which acts as an ordering identifier within that partition in order to allow consumers to start and stop reading from an arbitrary position.

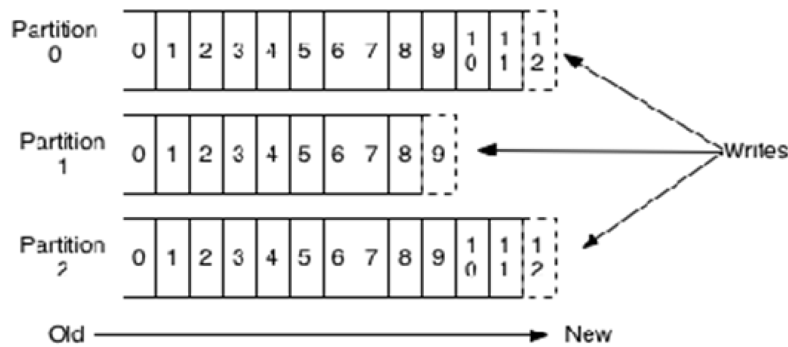

Figure 1: A Kafka Topic with 3 partitions with new records being added.

2.5 Replication and fault-tolerance

Kafka clusters also provide redundancy mechanisms that allow brokers to keep backups and be able to recover from failures. The unique combination of a record's topic, partition id and offset are what allows brokers to correctly replicate partitions. In that case one broker becomes the leader and the rest of the brokers with the replicated partitions become followers. In case of failure a new leader is elected from an "in-sync" subset of brokers (caught up with the leader's log) and seamlessly adopts the active consumers of the partition. The reason a new leader is elected only by in-sync replicas is because it's necessary for all recently committed messages to also be available on the new leader. It's also worth mentioning that message is considered committed only if it has been successfully copied to all in-sync replicas.

2.6 Producers

Kafka, as a Pub/Sub system, consists of data producers and consumers that read and write key-value records to the available distributed queues. Producers send key-value pairs to a specific topic but are usually oblivious as to which partition received the records since by default

the message will be sent to a random partition in order to load balance incoming traffic. However, all records that share the same key will end up in the same partition, unless of course that key is null in which case the message will be sent to a random partition. The process of assigning a set of keys to a specific partition is done by the Kafka practitioner, a process that maps messages to partitions, which by default simply hashes the key based on the number of available partitions.

2.7 Consumers

A Kafka consumer subscribes to one or more topics and can read messages from one or more topic partitions but a consumer instance is required for every topic partition. Each record in a partition contains an offset field which is a unique and monotonically increasing integer that's used by the consumer to keep track of how much they have advanced in that specific topic. Consumers can group together and form consumer groups, a process in which multiple consumers read from a single topic and each partition is only consumed by a single member of the group. In case of a consumer failure the remaining members of the group will rebalance the partitions of the failed consumer.

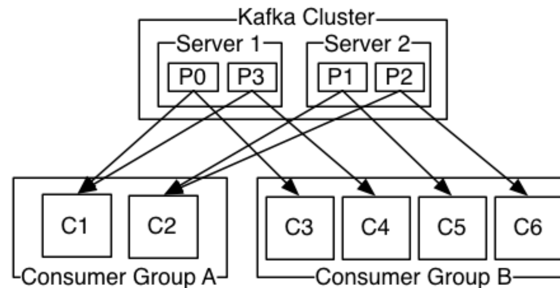

Figure 2: Two Kafka producers with two uneven consumer groups.

2.8 Kafka Streams

Kafka Streams simplifies development by building on the Kafka producer and consumer APIs while leveraging the capabilities of Kafka to offer data parallelism, coordination and fault tolerance. By applying stream processing techniques, a Kafka Streams application can seamlessly subscribe to a Kafka topic, consume the incoming stream and perform various stream transformations similar to those found in traditional stream processing frameworks like grouping, windowing, aggregations, joins and even custom transformations.

2.8.1 Architecture

Kafka Streams uses the concepts of partitions and tasks as units of its parallelism model based on Kafka topic partitions. More specifically each stream partition is a totally ordered sequence of data records and maps to a Kafka topic partition. A data record in the stream corresponds to a Kafka message from that topic. The data record keys determine the partitioning of data which is how data is routed to specific partitions within topics. A processor topology is scaled by breaking it into multiple tasks. More specifically, Kafka Streams creates a number of

tasks based on the number of input stream partitions with each task assigned a list of partitions from the input topics.

The user can configure the number of threads used to parallelize processing within an application instance and each thread can execute tasks and their topologies independently. Starting more stream threads or more instances of the application means that the topology will be replicated and process a different subset of Kafka partitions, effectively parallelizing processing. It is worth noting that there is no shared state between threads, therefore no inter-thread coordination is necessary.

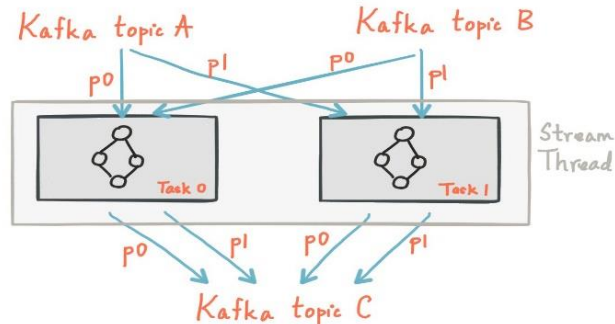

Figure 3: A stream with two topics and two stream tasks.

2.8.2 State stores

Kafka Streams provides state stores, which can be used by stream processing applications to store and query data which is important capability when dealing with stateful operations. The Kafka Streams DSL automatically creates and manages such state stores when stateful operators such as join() or aggregate() or windowing are called. Every stream task in a Kafka Streams application can contain one or more local state stores that can be accessed via Kafka Streams APIs in order to store and query data. Kafka Streams offers fault-tolerance and automatic recovery for local state stores.

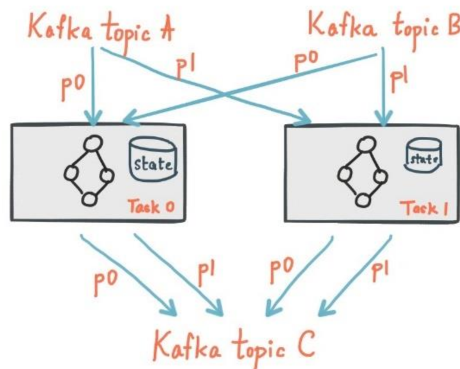

Figure 4: Two stream tasks with their dedicated local state stores.

3. Stream processing

Batch processing frameworks like Hadoop MapReduce and Apache Spark are still widely adopted by businesses around the world as Data Analytics tools due to their high throughput, fault tolerant and easily scalable design. However, the demand for low latency and more complex computations is higher than ever as the need for real time results, data preservation and high result accuracy is increasing rapidly. In order to fill these demands various stream processing frameworks have been created that require less costly infrastructure, operate on per-tuple processing semantics, have stronger processing guarantees and produce more accurate and faster results.

Stream processing frameworks also support different notions of time for their incoming and outgoing tuples. Processing time is the "classic" notion used by most older frameworks where the incoming tuple is assigned the timestamp of the system clock in each operator, a simple solution that requires no coordination or synchronization between cluster nodes but can lead to non-deterministic results as late arrivals or out-of-order data isn't taken into account at all. Event time on the other hand, aims to solve these issues by assigning each tuple the timestamp of its creation time (essentially the time when the event was produced by a device). Although this approach enables a framework to produce consistent results while handling late and out-of-order data the user must specify how the framework should extract the timestamp from each incoming tuple. Finally, there is also the notion of the ingestion time where each tuple is assigned a timestamp based on the time it entered the frameworks pipeline, a solution that doesn't require further configuration but still doesn't handle correctly out-of-order or late data.

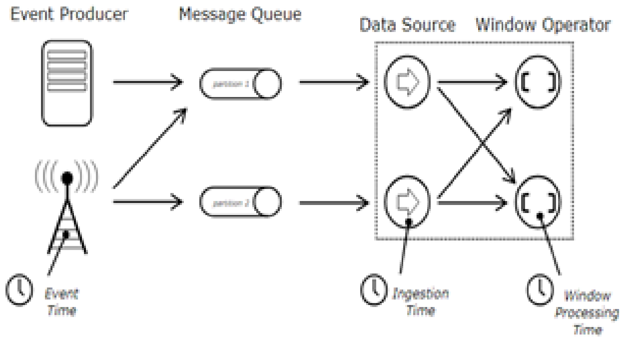

Figure 5: Event, Ingestion and Processing times.

Windowing mechanisms are also an integral part of every stream processing framework as incoming unbounded data sometimes needs to be divided into buckers and processed as a whole before producing any results. The most simple and easier to understand type of windows is the Tumbling (or Fixed) window where incoming data is partitioned into buckets that don't overlap every period and emitted at the end of each window as a set of tuples to the downstream operators. A window with a fixed period but also a sliding period is called a Sliding Window with the main difference being that on every sliding period a fixed window is applied to the incoming

data. Finally, Session Windows partition incoming data into unequal sized buckets after a set amount of time has passed between two events, this fixed amount of time is called a session gap.

Last, but not least, most streaming frameworks work out-of-the-box with various I/O connectors for distributed message queues and file systems designed specifically for streaming use cases. Compatibility with existing middleware like Kafka, ZeroMQ and RabitMQ for example is very important as they allow developers to focus on the core logic of their programs without the need to implement such connectors.

3.1 Apache Beam

Apache Beam is an open source, unified model for defining both batch and streaming data-parallel processing pipelines which can then be executed locally or by one of Beam's supported distributed processing back-ends. Beam's strength can be easily seen when processing Embarrassingly Parallel data processing tasks, in which the initial problem can be quickly decomposed into many smaller data bundles that will then be processed independently, and most importantly, in parallel. Beam can also be used for Extract, Transform, and Load (ETL) tasks and pure data integration, useful for moving data between different storage media and data sources, transforming data into a more desirable format or even loading data onto a new system.

3.1.1 Overview

Apache Beam provides a unified programming model that can represent and transform data sets of any size, whether they are produced by a bounded or an unbounded source [2]. It's also worth noting that Beam uses the same classes to represent both bounded and unbounded data as well as the same transforms to operate on that data. By using transformations Beam can effectively and quickly read, process and save data, from and to, various distributed data sources and sinks. Finally, the operators responsible for such transformations form a Beam Pipeline which usually starts and ends with a distributed data source or sink. The Apache Beam Pipeline Runners translate the data processing pipeline defined by the user into an API compatible with the distributed processing back-end of the user's choice. We should also mention that Beam provides the tools to build custom runners that can be executed on any platform capable of supporting a JVM provided that they implement various APIs ranging from the creation of PCollections to fault-tolerant semantics. Although our experiments focused on Apache Spark [3], Apache Flink [4] and Apache Apex, Beam currently supports out of the box Runners that work with the following distributed processing back-ends:

- Apache Apex
- Apache Flink
- Apache Gearpump
- Apache Samza
- Apache Spark
- Google Cloud Dataflow
- Direct runner (for debugging purposes)

Beam provides a number of abstractions that simplify the mechanics of large-scale distributed data processing for both batch and streaming data sources. A very important one is the Pipeline, which encapsulates the entire data processing task which starts with reading input data, transforming it and finally writing output data. A pipeline can also take extra arguments either to enrich user code or specify runner specific parameters (e.g. Memory options). However, runners can choose how to implement certain pipeline details in order to optimize things like transformation chains, I/O from distributed sources and even switch operator order in order to reduce data shuffling.

3.1.2 Parallel Collections

A Parallel Collection (PCollection) represents a potentially distributed, multi-element data set that a Beam pipeline operates on as Beam transforms use PCollections as inputs and outputs. Input data can originate from a static source like a text file, a continuously updating source or an in-memory relation inside the driver. External data sources require the use of a Beam-provided I/O adapter which connects to a file system in a fault-tolerant way and retrieves a collection of tuples which can then be transformed into a PCollection. Beam provides the tools for users to build their own I/O adapter but there is a wide variety of already implemented ones that work out of the box and support popular file-systems and message queues like HDFS, Apache Kafka, Zero-MQ and S3.

PCollections are similar to some distributed collections found in other programming languages have some unique characteristics. To start with, PCollections can't be shared or reused between different pipelines and their elements must all share the same serializable type, whoever that may be. Moreover, a PCollection is immutable and once created can't no elements can be added or modified, thus a transformation is the only way to process a PCollection. The accessibility of PCollection elements is also quite limited since random access is prohibited, instead a transformation considers every element individually. Furthermore, there is no upper or lower bound in a PCollection's size whether it can fit inside the driver or needs to be distributed to remote machines. A PCollection can also be bounded or unbounded in size, meaning

3.1.3 Transformations

Transforms are the operations a pipeline executes in order to process input data by using function objects to process each element of one or more input PCollections. Depending on the pipeline runner, back-end and transform logic many different workers across a cluster may process the input elements. In such case, the partial transform being executed on each worker generates a set of output elements that are ultimately added to the final output PCollection that the transform produces. To invoke a transform, one must apply it to an input PCollection. Invoking multiple Beam transforms is similar to method chaining like the following example:

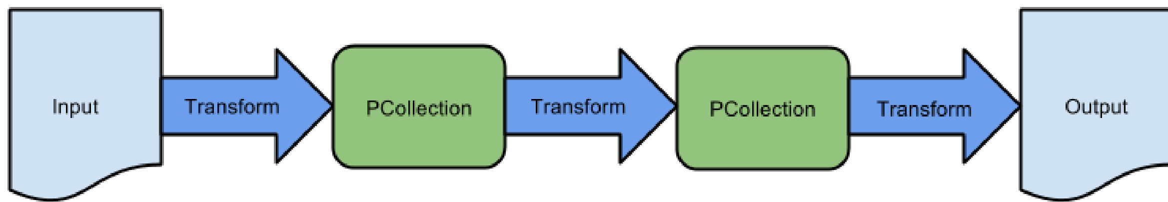

Figure 6: Transform chaining of an input parallel collection.

The order of pipeline's transforms determines the structure of the pipeline since the best way for someone to think of a pipeline is as a directed acyclic graph (DAG), where the nodes are PCollections and the edges are transforms. Beam provides some core transforms, each of which represents a different processing paradigm, but custom or composite transformations can also be created.

ParDo (Parallel Do) is a Beam transform for generic parallel processing. The ParDo processing paradigm is similar to the Map phase of a standard Map-Reduce algorithm since it reads each element in the input PCollection, then performs some processing function on that element and finally emits zero or more elements to an output PCollection. ParDo is useful for a variety of data processing operations like filtering a data set as one can use ParDo to check each element in a PCollection and either output that element or discard it. ParDo can also be used for formatting or type-converting elements in a data set since if the input PCollection contains elements that are in a different format ParDo can be used to perform a conversion on each element and output the result to a new PCollection. Extracting parts of each element in a data set is also possible with ParDo. If there is a PCollection of records with multiple fields, for example, ParDo can be used to parse out just the necessary fields and create a new PCollection. Finally, ParDo is also used to perform computations on each element in a data set. ParDo can be used to perform simple or complex computations on every element, or certain elements, of a PCollection and output the results as a new PCollection.

When applying a ParDo transform user code is provided in the form of a Do Function (DoFn) object which is a Beam SDK class that defines a distributed processing function. The DoFn object that is passed to ParDo contains the logic that gets applied to the elements in the input parallel collection. If a ParDo performs a '1-1' mapping of input elements to output elements the use of the higher-level MapElements transform, which can accept an anonymous lambda function as well, is recommended.

GroupByKey is a Beam transform for processing collections of key-value pairs (KVPs). As a parallel reduction operation, analogous to the Shuffle phase of a Map-Reduce algorithms, the input to GroupByKey is a collection of KVPs that represents a multimap, a collection that contains multiple pairs with the same key. Given such a collection, one can use GroupByKey to collect all of the values associated with each unique key in order to aggregate data that has something in common.

When using unbounded PCollections, one must use either non-global windowing or an aggregation trigger (more in the following section) in order to perform a GroupByKey or CoGroupByKey. This is because a bounded GroupByKey or CoGroupByKey must wait for all the data with a certain key to be collected but with unbounded collections the data is unlimited and windowing should allow grouping to operate on logical, finite bundles of data within these unbounded data streams. Finally, when grouping by key all of the PCollections that need to be grouped must use the same windowing strategy and window sizing in order to avoid conflicts.

CoGroupByKey performs a relational join of two or more key-value PCollections that have the same key type. Using CoGroupByKey should be considered when multiple data sets that provide information about related things need to be joined. When using unbounded PCollections, one must use either non-global windowing or an aggregation trigger in order to perform a CoGroupByKey. CoGroupByKey accepts multiple PCollections as input and produces a KVP for each unique element in the key sets of the input PCollection tuples with an iterator of all tuples that share the same key as a value. Finally, it's worth mentioning that relational joins of any kind can be implemented using a CoGroupByKey transform (including multi-way relational joins).

Combine is a Beam transform for combining collections of elements or values in your input data. Combine has variants that work on PCollections, and some that combine the values for each key in PCollections of KVPs. When applying a Combine transform, one must provide the function that contains the logic for combining the elements or values. That function should be commutative and associative, as the function is not necessarily invoked exactly once on all values with a given key. Because the input data may be distributed across multiple workers, the combining function might be called multiple times to perform partial combining on subsets of the value collection. Apache Beam also provides some pre-built combine functions for common numeric combination operations such as sum, min, and max.

Flatten is a Beam transform for PCollection objects that store the same data type as Flatten merges multiple PCollection objects into a single list of logical PCollections. When using Flatten to merge PCollection objects that have a windowing strategy applied, all of the PCollection objects must use a compatible windowing strategy and window sizing. This usually boils down to the windows having the same length, step and trigger.

Partition is a Beam transform for PCollection objects that store the same data type. Partition splits a single PCollection into a fixed number of smaller collections by dividing the elements of a PCollection according to a user-provided partitioning function that the user provides. The partitioning function contains the logic that determines how to split up the elements of the input PCollection into each resulting partition PCollection. The number of partitions must be determined at graph construction time which is usually done by passing the number of partitions as a command-line option at runtime.

3.1.4 Beam I/O

Apache Beam provides some extra tools in its API in order to allow for more complex pipelines and user-friendly design.

Side inputs are an addition to the main input PCollection of a ParDo transform. A side input is an additional input that a DoFn can access each time it processes an element in the input PCollection. When someone specifies a side input, a view of some other data is created that can then be read from within the ParDo transform's DoFn while processing each element. Side inputs are extremely useful if a ParDo needs to inject additional data when processing each element in the input PCollection, but the additional data needs to be determined at runtime. Such values might be determined by the input data, or depend on a different branch of the pipeline.

While ParDo always produces a main output PCollection they can also produce any number of additional output PCollections. When a ParDo has multiple outputs, it returns all of the output PCollections, including the main output, bundled together (similar to a Flatten transform result).

When creating a pipeline, it is often needed to read or write data from some external source, such as a remote distributed database. Beam provides read and write transforms for a number of common data storage types but also allows users to implement their own read and write transforms. Read transforms read data from a remote source and return a PCollection of the data while write transforms write the data of a PCollection to an external data source. Some common I/O adapters that work out of the box with Apache Beam are the ones for HDFS, S3 and Kafka.

3.1.5 Windowing

Windowing is a technique used to subdivide a PCollection according to the timestamps of its individual elements. Transforms that aggregate multiple elements, such as GroupByKey and Combine process each PCollection as a succession of multiple, finite windows, though the entire collection itself may be of unbounded size. However, when working with an unbounded data set, it is impossible to collect all of the elements, since new elements are constantly being added and may be infinitely many so when working with unbounded PCollections, windowing is especially useful.

In the Beam model, any PCollection can be subdivided into logical windows. Each tuple 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. However, Beam's default windowing behavior is to assign all elements of a PCollection to a single, global window and discard late data, even for unbounded PCollections. Before using a group transform such as GroupByKey on an unbounded PCollection, one must perform some extra tasks like setting a non-global windowing function and a non-default trigger which in turn allows the global window to emit results under other conditions, since the default windowing behavior will never occur.

After setting the windowing function for a PCollection, the elements' windows are used the next time grouping transform is applied to that PCollection. Window grouping occurs on an as-needed basis. If a windowing function is set using the Window transform, each element is assigned to a window, but the windows are not considered until GroupByKey or Combine aggregates across a window and key.

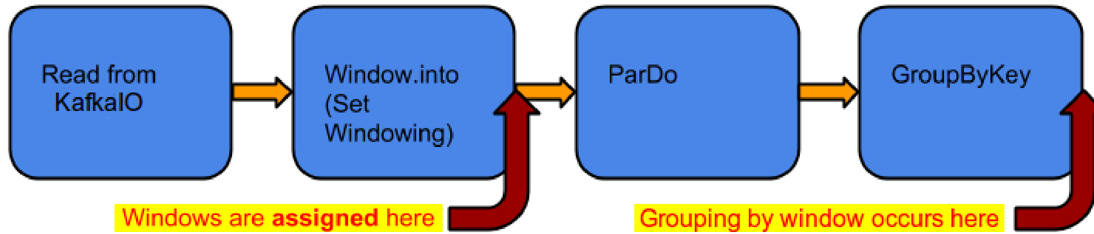

Figure 7: Windowing and grouping in succession.

In the above pipeline, an unbounded PCollection is created by reading a set of key/value pairs using KafkaIO and then a windowing function is applied to that collection using the Window transform. A ParDo is then applied to the collection and then later the result of that ParDo is grouped using GroupByKey. The windowing function has no effect on the ParDo transform because the windows are not actually used until they're needed for the GroupByKey but subsequent transforms are still applied to the result of the GroupByKey.

Beam provides several windowing functions, including:

- Fixed Time Windows
- Sliding Time Windows
- Per-Session Windows
- Single Global Window
- Calendar-based Windows

Fixed time windows are the simplest form of windowing. Using fixed time windows means that given a timestamped PCollection each window will capture all elements with timestamps that fall into a specified interval, usually defined by some time duration. A fixed time window represents a sequence of non-overlapping time intervals in the input data stream whose elements are bundled into parallel collections and emitted by default at the end of the time interval.

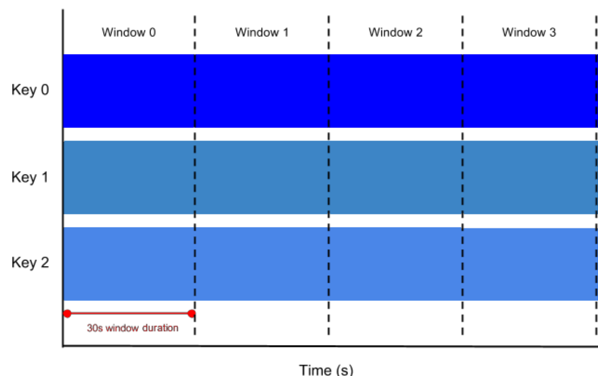

Figure 8: Example of fixed time windows on three different streams.

Sliding time windows represent time intervals in the data stream, but unlike fixed-time windows, they can overlap. More specifically each window captures a set amount of data, usually specified by a time duration, but at the end of that duration the window is offset by a fixed amount of time which is the window period. Because multiple windows can overlap, most elements in a data set will belong to more than one window.

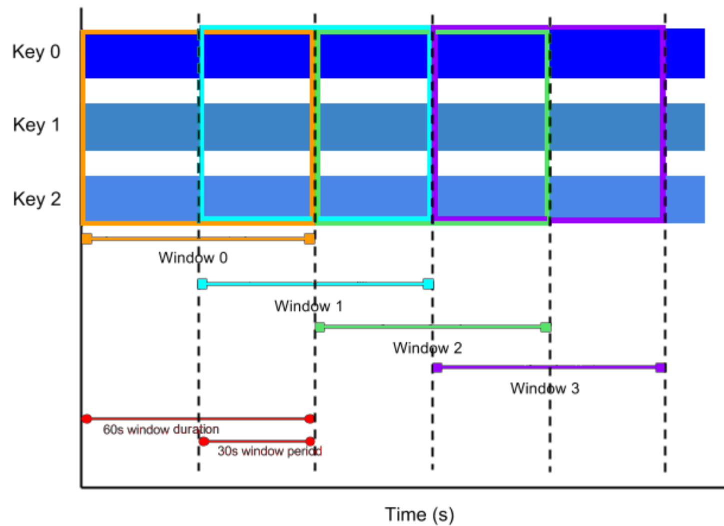

Figure 9: Fixed time windows.

Session windows partition input elements in buckets based on a certain gap duration between groups of elements. Session windowing applies on a per-key basis and is useful for data that is irregularly distributed with respect to time. If data arrives after the minimum specified gap duration time the start of a new window is initiated. This is extremely useful when measuring user activity or want to produce as few windows as possible by creating a window after a certain time has passed.

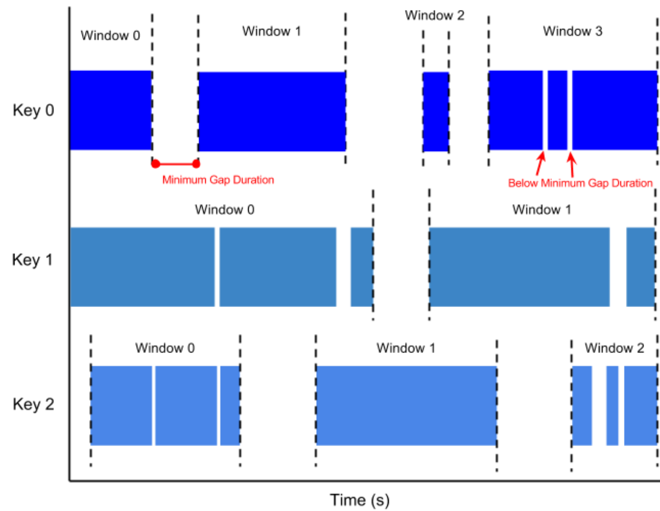

Figure 10: Example of a session time window.

The single **global window** is used by default, as all data in a PCollection is assigned to that window, and late data is discarded. If user data set is of a fixed size, they can use the global window default for their PCollection. 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, users should specify a non-default trigger for that PCollection. Users are advised to be cautious when using the global windowing strategy since, unlike computer memory, streams may be infinite.

3.1.6 Watermarks and Triggers

When collecting and grouping data into windows triggers are used in order to determine when to emit the aggregated results of each window, usually referred to as a pane. Using Beam's default windowing configuration and default trigger, Beam outputs the aggregated result when it estimates all data has arrived and discards all subsequent data for that window. The user can set triggers for their PCollections to change this default behavior.

Beam provides a number of pre-built triggers that users can set.

- Event time triggers are Beam`s default triggers and operate on the event time, as indicated by the timestamp on each data element.
- Processing time triggers operate on the processing time which is the time when the data element is processed at any given stage in the pipeline.
- Data-driven triggers operate by examining the data as it arrives in each window, and firing when that data meets a certain property.
- Composite triggers combine multiple triggers in various ways.

Triggers provide two additional capabilities compared to simply outputting at the end of a window. Firstly, Beam is allowed to emit early results, before all the data in a given window has arrived which is useful for speculative results. Triggers also allow processing of late data by triggering after the event time watermark passes the end of the window. These capabilities allow for better control of data and balance between different factors depending on the use case. One can also set a trigger for an unbounded PCollection that uses a single global window for its windowing function which can be useful when the pipeline needs to provide periodic updates on an unbounded data set.

Event time triggers operates on event time and more specifically the AfterWatermark trigger emits the contents of a window after the watermark passes the end of the window based on the attached timestamps, which are a global progress metric, and is Beam's notion of input completeness within the pipeline at any given point. In addition, users can configure triggers that fire if the pipeline receives data before or after the end of the window.

Processing time triggers operate on processing time and emit a window after a certain amount of processing time has passed since data was received. The processing time is determined by the system clock, rather than the data element's timestamp. There is also the ability for triggering early results from a window, particularly a window with a large time frame such as a single global window.

Data-driven triggers provide one data-driven trigger which works on an element count which means it fires after the current pane has collected at least N elements. This allows a window to emit early results, which can be particularly useful when using a single global window.
It is important to note that if the number of elements don't arrive, those elements will sit around forever, therefore consider the use of a composite trigger to combine multiple conditions. This allows for multiple firing conditions such as the arrival of a set number of elements or after some time.

Finally, composite triggers allow the user to combine multiple triggers in order to form composite triggers, and can specify a trigger to emit results repeatedly, at most once, or under other custom conditions. Additional early or late firings can be added as well as a repeating trigger which executes forever and any time the trigger's conditions are met, it causes a window to emit results and then resets and starts over.

Keywords like AfterFirst, AfterAll and orFinally can be used in order to control the trigger firing flow. AfterFirst takes multiple triggers and emits the first time any of its argument triggers is satisfied which is equivalent to a logical OR operation for multiple triggers. AfterAll takes multiple triggers and emits when all of its argument triggers are satisfied. This is equivalent to a logical AND operation for multiple triggers. Lastly, orFinally can serve as a final condition to cause any trigger to fire one final time and never fire again.

Another important aspect of window triggering is the window accumulation mode which controls what part of the window contents are emitted when a trigger fires. Since a trigger can fire multiple times, the accumulation mode determines whether the system accumulates the window panes as the trigger fires or simply discards them after firing. The user can set a window to accumulate the panes that are produced when the trigger fires via accumulatingFiredPanes() when they set the trigger or discard the fired panes by invoking discardingFiredPanes() on the trigger. These modes are extremely important when using accumulators on windowed collections.

4. Platform implementation and metrics collection

In this section we present the structure of the proposed platform and the structure of the individual subsystems that our platform comprises of. More specifically we describe the process of collecting and analyzing metrics from various stream processing frameworks and how they are used to affect the decision making of queries. We also touch on the data preprocessing steps that take place in the Kafka Streams application, its topology and communication mechanisms with other parts of the platform. Finally, we present the Beam pipeline which was used to implement five join queries in order to test the effectiveness of our decision-making algorithm in different platforms by using Beam Runners. The goal of the proposed platform is to asses which queries should be sent to the available frameworks in order to optimize throughput and latency.

4.1 Kafka Streams topology

A Kafka Streams application is the first module of our pipeline and is mainly used for data preprocessing and routing. The topology of the application is quite simplistic but requires a high degree of parallelism with fault tolerant semantics and Kafka I/O adapters working out of the box, a use case that suites well Kafka Streams.

The ingestion process begins with a set of incoming data streams of key-value pairs entering the Kafka pipeline via a Kafka consumer, all of which are evenly load balanced across the application parallel instances. Incoming tuples are then forwarded to a Kafka Streams Transformer which keeps a local sample of tuple keys and immediately forwards the tuples to the correct Kafka Topic according to the active queries at the processing time. We should also mention that each beam runner has one dedicated topic for incoming tuples, therefore when a new query arrives and it is decided that a specific framework must execute that query, Kafka Streams simply routes incoming tuples to the corresponding topic. New queries are broadcasted periodically to the application by a different module of our platform which is discussed in the following section but the important think to note is that our Kafka Streams application reroutes incoming traffic based on the incoming queries.

The application also contains a very simplistic REST API which can be used by the parallel instances of the Kafka Streams application in order to exchange messages and requests. Since the transformer can sample the keys of the stream partition it has been assigned, we can use the REST API to combine the local samples from each transformer in order to produce a global sample of the incoming data streams which can later be used to extract useful information about the data skew and size. In order to achieve something like that we use a distributed sampling algorithm with a reservoir that takes into account the size of the input streams and works in 2 passes, it can be found in the following section.

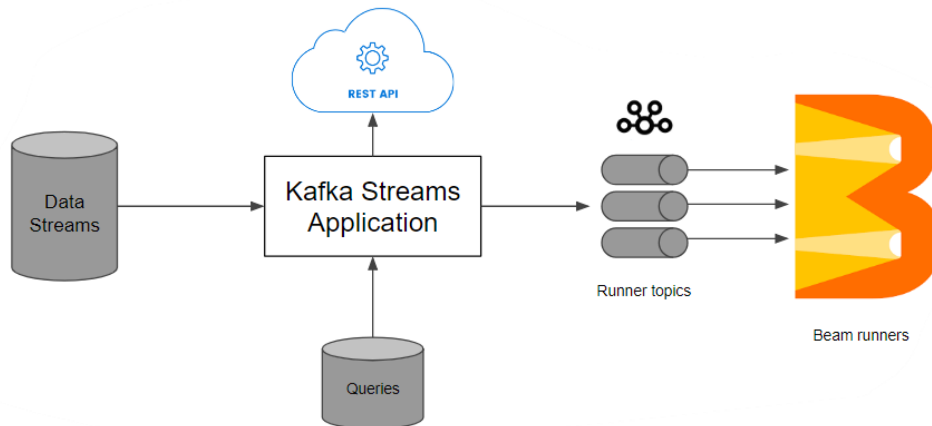

Figure 11: Kafka topics and the proposed Kafka Streams topology.

4.1.1 Reservoir sampling

Reservoir sampling is a family of randomized algorithms for randomly choosing a sample of k items from a list S of n items, where n is either very large or unknown. After the sampling, each item in the list should have equal probability of k/n being chosen. The original sampling algorithm with a reservoir is called algorithm R [5] and is as follows:

```
1: procedure ALGORITHM R(S[1..k], R[1..n])
2:     for i ← 1 to k do
3:         R[i] ← S[i]
4:     end for
5:     for i ← k + 1 to n do
6:         j ← random integer between 1 and i
7:         if j < k then
8:             R[j] ← S[i]
9:         end if
10:    end for
11:    return R[1..k]
12: end procedure
```

Figure 12: Algorithm R pseudocode.

In many applications the amount of data needed from a small sample is too large and it is desirable to distribute sampling tasks among many machines in parallel in order to speed up the process, therefore a parallel version of the algorithm R is necessary. Without loss of generality, let us assume there are two sub-streams of size m and n respectively where both m and n are greater than k.

In the first step of the algorithm, workers process their own sub-streams in parallel, using the standard algorithm R. When both workers finish their sub-stream traversal, two reservoir lists R and S are created. In addition, both workers count the number of items in their own sub-streams during the traversal, and thus m and n are known when R and S are available.

The following step required us to combine the two reservoir lists to get k items out of them. To achieve this, weights are assigned to items according to the sizes of their sub-stream and then proceed to a second sampling phase. The second phase requires k iterations for the sampling to complete, as many as the number of maximum items that will be in the final sample. Every iteration starts with generating a random number between 0 and 1 such that, with probability $p = m/(m+n)$, a random sample from reservoir list R is picked, and with probability $1-p$, a random sample from reservoir list S is picked instead. At the end of the kth iteration a final reservoir list for the entire stream has been created.

This algorithm is described as follows:

```
1:  procedure DISTRIBUTED ALGORITHM R(R[1..m], S[1..n], T[1..k])
2:      for sub-stream s : sub-streams do
3:          Perform individual reservoir samplings and count the length
    of S
4:      end for
5:      p ← m/(m+n)
6:      for i ← 1 to k do
7:          j ← random number between 0 and 1
8:          if j ≤ p then
9:              move a random item from R to T
10:         else
11:             move a random item from S to T
12:         end if
13:     end for
14:     return T[1..k]
15: end procedure
```

Figure 13: Distributed Algorithm R pseudocode.

In each iteration of the second phase any item in the entire stream has probability of $1/(m+n)$ being chosen, therefore any item has probability of $k/(m+n)$ being chosen and thus the algorithm generates k random samples. The algorithm runs in $O(max(m,n))$ time since the algorithm can pick at most $max(m,n)$ items at random from the two lists. The space complexity is $O(k)$ because only a sample of at most k items is kept in-memory at any time.

Implementing this distributed algorithm in a Kafka Streams application is quite simple as the number of parallel workers is defined by the number of application parallel instances and the stream samples and their respective sizes can be sent via the REST API of each instance. We should also mention that each parallel instance is aware of other parallel instances and can retrieve their IP and port on demand. At the moment the partial samples are gathered on the first parallel instance where the second phase of the distributed algorithm takes place. Finally, the sample can be made publicly available with the REST API for other parts of our platform to use.

4.2 Statistics and query assignment

In order for our platform to effectively assign queries to available frameworks a wide variety of statistics and available metrics is taken into account. The Controller module is assigned that task along with deciding what query each available framework will execute. We will now briefly explain the metrics and statistics the Controller module is capable of collecting.

4.2.1 Metrics and statistics acquisition

To start with, most stream processing frameworks expose a REST API with some available processing metrics which the user can query via GET and POST requests. Furthermore, most stream processing frameworks also provide statistics for their operators as well as their health, work load and processed tuples. Finally, resource managers (RMs) like YARN can also provide useful information about the available frameworks usually regarding their allocated resources like memory, available cores, CPU clocks and uptime. The full list of metrics provided by the frameworks and RMs is the following:

- Apache Spark Master
  - Job start, elapsed and end time.
  - Number of Job attempts and current status.
  - Statistics for each batch:
    - ID and status.
    - Start, elapsed and finish time.
    - Input size.
    - Active and completed output Ops.
    - Scheduling, processing and total delay.
  - Statistics for each executor:
    - ID, port and activity status.
    - Number of CPU cores, RAM size and disk usage.
    - Number of active tasks and RDD blocks.
    - GC time.
    - Total Input, output and Shuffle R/W bytes.
    - Total/Used on/off heap memory.
  - Statistics for each Stage:
    - ID and status.
    - Number of active tasks.
    - Input and output bytes.
    - Shuffle R/W bytes and R/W records.
    - Bytes spilled in disk.
    - Names and values of accumulators.
  - Statistics for each Streaming Receiver:
    - Start, elapsed and finish time.
    - Batch duration.
    - Number of active, inactive, total, completed receivers.

        - Number of active and completed batches.
        - Average scheduling, processing and total delay time.
- Apache Flink Job Manager
    - Job id, name and status.
    - Start, elapsed and finish time.
    - Statistics for each Job Plan Node (per stream operator):
        - Node ID, status and operator name.
        - Parallelism.
        - Inputs, outputs and shipping strategy.
        - Number of R/W records and bytes.
        - Health and Backpressure status.
        - Number of tasks.
        - Accumulator values.
        - Statistics for each operator task:
            - Record throughput (avg, min, max, 75th and 99th percentiles)
            - Byte throughput (avg, min, max, 75th and 99th percentiles)
            - Record latency (avg, min, max, 75th and 99th percentiles)
- YARN Master
    - Application start, elapsed and finished time.
    - Application progress, priority and requested resources.
    - Container memory size, cores and virtual cores.
    - Node CPU clocks, memory size, running application and its requested resources.
    - UI address.
    - User id, name and history.

Although RMs and stream processing frameworks can provide a variety of useful metrics sometimes they are not enough. Our platform produces also provides some metrics data normality metrics generated from the Kafka Streams sampling routines as discussed in the previous section. More specifically, the skewness and kurtosis of incoming tuple keys are calculated and taken into account when assigning queries.

In statistics, skewness is a measure of the asymmetry of the probability distribution of a random variable about its mean. The skewness value can be positive, negative, or even undefined and f skewness is 0 the data are perfectly symmetrical.

$$\frac{\frac{1}{n}\sum_{i=1}^{n}(x_i-\overline{x})^3}{\left(\frac{1}{n}\sum_{i=1}^{n}(x_i-\overline{x})^2\right)^{3/2}}$$

Figure 14: Data skew of random variable x

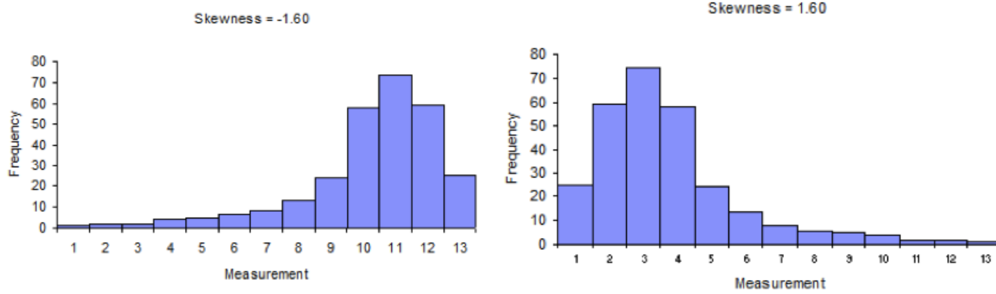

Figure 15: Examples of skewed distributions.

Kurtosis provides information about the height and sharpness of the central peak, relative to that of a standard bell curve.

$$\frac{\frac{1}{n}\sum_{i=1}^{n}(x_i-\overline{x})^4}{\left(\frac{1}{n}\sum_{i=1}^{n}(x_i-\overline{x})^2\right)^2}-3$$

Figure 16: Kurtosis of random variable x.

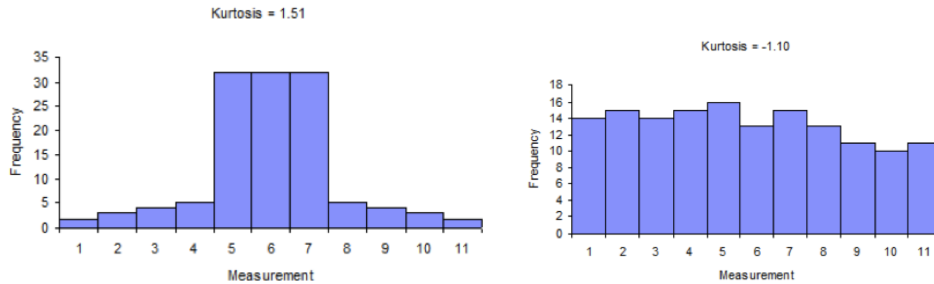

Figure 17: Different values of Kurtosis.

Finally, Apache Kafka and Apache beam both provide metrics related to their work-load like tuple latency and throughput. However, Beam relies on the underlying runner to provide available metrics to the user meaning that some runners can't provide pipeline metrics in real-time and sometimes such metrics are not supported by the runner entirely.

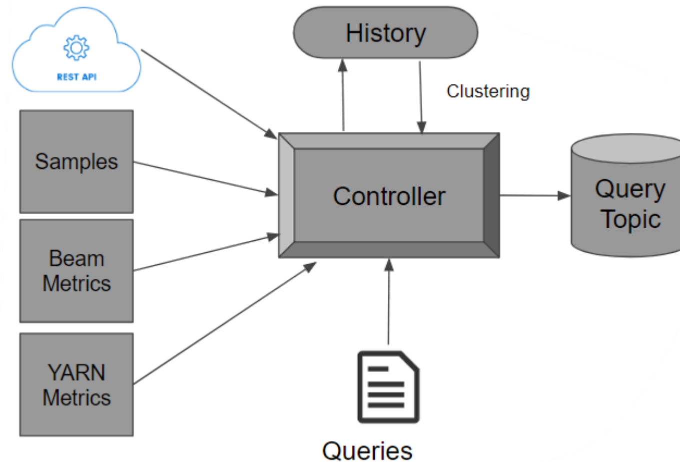

Figure 18: Overview of the Controller sub-module.

4.2.2 Query assignment

After gathering various statistics and metrics our proposed platform is capable assigning queries to available frameworks based on a strategy. Although there are many ways to assign queries to frameworks, especially given the volume of available metrics, we propose and evaluate three.

Firstly, the Round Robin strategy simply assigns incoming queries to frameworks in a round robin fashion without taking into consideration any of the available metrics. The second proposed strategy consists of every query being sent to the Apache Flink framework, again without considering any available metric. Finally, a strategy which considers the value of every metric collected by our platform and also takes into account previously executed queries is the LSH strategy. LSH stands for Locality Sensitive Hashing [6] which is a technique that allows one to find similar entries in large datasets and has been successfully applied to various fields such as biology, weather forecasting, video editing and more. This strategy aims to match new queries to frameworks that have previously executed the same queries while having similar metrics and statistics (e.g. similar skewness and latency). This strategy, however, requires us to keep a list of previously executed queries and their respective statistics.

LSH is based on the idea that if two multi-dimensional points are close together then after a "projection" operation these two points will remain close together. More specifically, a random projection maps a data point from a high-dimensional space to a lower-dimensional subspace. The projection operation can be a simple scalar projection given by $h(\vec{u}) = \vec{u} \cdot \vec{x}$ where $\vec{u}$ is a high dimensional vector and $\vec{x}$ is a vector with components selected from the random Gaussian distribution $\sim N\ (0{,}1)$. This scalar projection is then quantized into a set of hash bins, with the intention that nearby items in the original space will fall into the same bin.

The resulting full hash function is:

$$h^{x,b}(\vec{u}) = \left\lfloor \frac{\vec{x} \cdot \vec{u} + b}{w} \right\rfloor$$

In the previous formula w is the width of each quantization bin and b is a uniformly distributed variable between 0 and w.

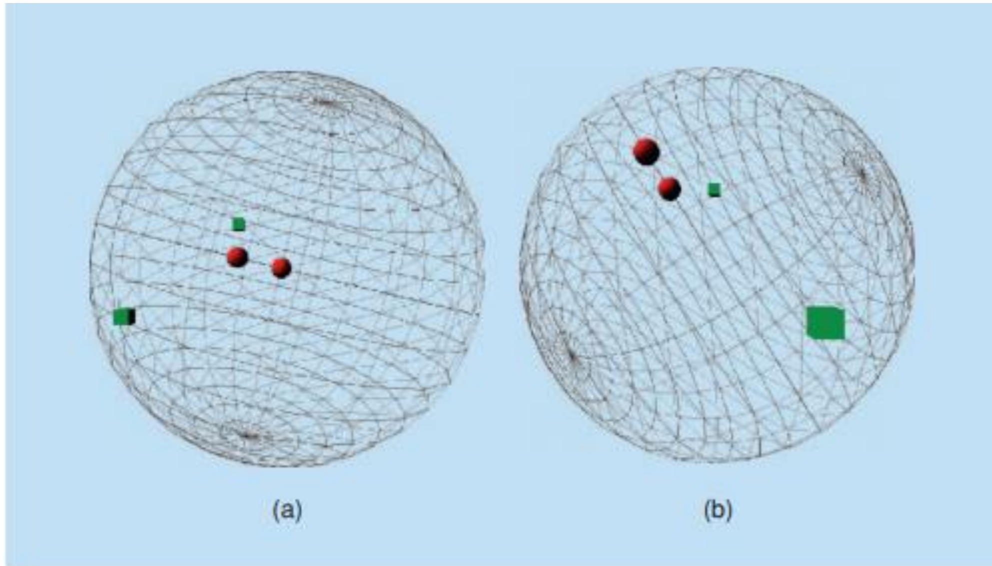

Figure 19: Projections of 2 close circles and 2 distant squares projected in this paper surface.

In order to implement LSH we need to convert the list of statistics for each previous query to a multi-dimensional vector with each statistic occupying exactly one component of this vector. Furthermore, frameworks can have different vector sizes since most some frameworks provide more statistics than others, therefore each framework must keep a list of their past queries and their respective metrics. When a new query arrives, the Controller module uses LSH to perform a k-NN search of the current query with a vector of the current statistics in every framework list in order to retrieve similar queries. If more than one results are returned our proposed strategy dictates that we pick the one that has the minimum latency.

4.3 Apache Beam pipeline and Runners

In order to test the effectiveness of our query assignment strategies we created a simple pipeline on Apache Beam which can execute 5 similar queries. The pipeline begins by reading data from two Kafka topics that contain tuples from two different data streams. Incoming tuples contain key-value pairs that consist of the join key, their respective values and a list of queries this tuple belongs in. We then split this tuple into multiple ones based on the number of its query IDs and create a composite key on each one that consists of its query ID and the original join key followed by the original tuple value. The following figure provides a good example.

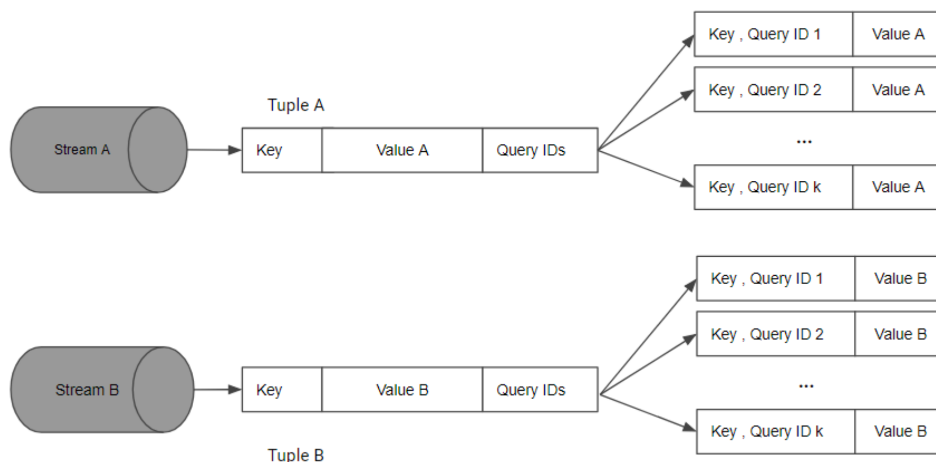

Figure 20: Extraction of query IDs.

After that, the two data stream results are windowed using the same window function in order to produce finite batches of data. After that a CoGroupByKey (CGBK) transform takes place that groups all key-value pairs with the same composite key together with a DoFn that provides access to all sets of values sharing the same composite key, thus simplifying the process of implementing the JOIN queries tremendously. After isolating the set of values per unique composite key we apply a Join algorithm based on the query ID derived from the composite key. Finally, after the correct query is executed the key-value pairs are emitted to a Kafka sink.

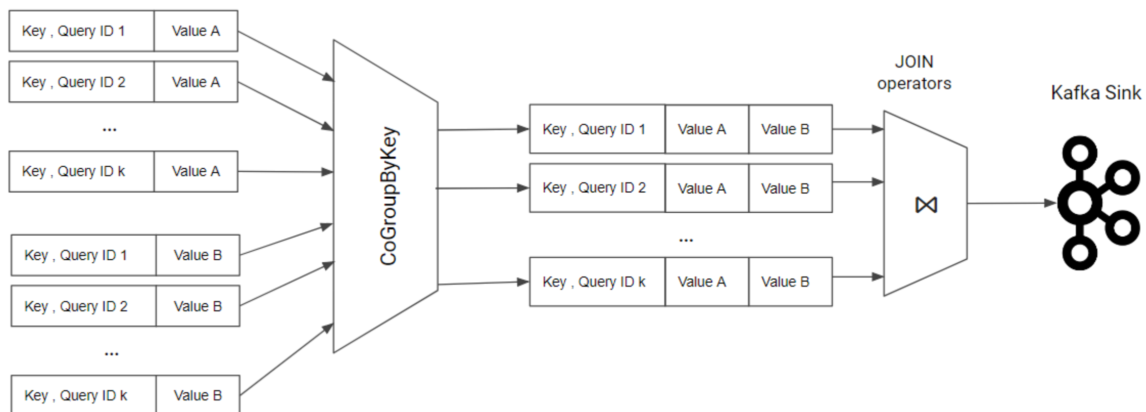

Figure 21: CGBK and Join operations.

Information about the number of tuples and their time spent inside our platform is also sent to the Controller module via an intermediate Kafka topic periodically. The following Join Algorithms where implemented in the Beam pipeline and perform a relational join between the relations R and S:

Inner Join

```
For each tuple r in R do
   For each tuple s in S do
      If r.key = s.key:
         output the tuple <r,s>
```

Full Outer Join

```
If S is empty:
   For each tuple r in R do
      output the tuple <r,null>
Else if R is empty:
   For each tuple s in S do
      output the tuple <null,s>
Else:
    For each tuple r in R do
      For each tuple s in S do
          If r.key = s.key:
             output the tuple <r,s>
```

Stream Semi Join

```
For each tuple r in R do
  For each tuple s in S do
     If r.key = s.key:
        output the tuple <r,null>
```

Left Outer Join

```
For each tuple r in R do
  For each tuple s in S do
     If r.key = s.key:
        output the tuple <r,s>
     Else:
        output the tuple <r,null>
```

Right Outer Join

```
For each tuple s in S do
   For each tuple r in R do
      If r.key = s.key:
         output the tuple <r,s>
      Else
         output the tuple <null,s>
```

5. Experimental Evaluation

In this chapter we present the results of our experiments and evaluate the performance of the Apache Beam Runners that were used. The framework group consists of Apache Spark, Apache Flink and Apache Apex, three frameworks that can handle streaming workloads under Apache Beam dataflow [7] pipelines. We start by analyzing the settings and parameters that were used to create a small cluster for each framework with the assistance of YARN and Flink's Job Manager. Moreover, we describe how our Kafka Streams application was designed and deployed in order to handle the input data streams without problems.

Our distributed experiments provide results for throughput and latency for many scenarios that involve two data streams with tuples from the public dataset of Fares and Taxis [8]. The first scenario tests the scalability of our Kafka Streams' application latency for various numbers of parallel instances. Scalability, experiments are also carried out for each framework (Spark, Flink and Apex) and provide results for the latency and throughput of each framework for varying degrees of parallelism and input stream rates.

Finally, throughput and latency experimental results are shown for each strategy for constant degree of parallelism and varying input stream rates. This group of experiments can help us draw conclusions regarding the effectiveness of our proposed strategies and how well each framework performed.

5.1 Cluster Setup

In order to run our experiments a small cluster for each framework has to be created. The high-level API of Apache Beam [9] generates a pipeline that can be ran n Apache Flink, Apache Spark and Apache Apex [10] as well as pass extra arguments to the underlying distributed backends like memory size and the number of CPUs to use. Apache Flink has its own Job Manager [11] that controls the number of task-slots each application receives in order to execute its pipeline. Apache Spark and Yarn use YARN [12] as a resource manager and their configuration settings go through the YARN master before any container is started. All of the Flink, Spark and Apex configuration properties (like memory size) are provided through the Beam high-level API as program arguments.

Regarding the framework-specific configurations it's important to mention that in order for Apache Spark (in streaming mode) to achieve a stable rate of incoming and outgoing records the batch size was increased to 5 seconds. Furthermore, the time spent reading incoming data in the pipeline sources were increased to 20% of the total batch time and the Spark cache was disabled entirely. Finally, the backpressure mechanisms were activated in order to stabilize the latency fluctuations when under high load. The reason we had to intervene and change some of Spark's internals was because the Kafka consumer and producers were performing poorly with the default settings which is expected since Apache Spark is not a pure streaming platform.

During the experiments a predefined set of queries was executed. Although that set was randomly generated with each query having an equal probability of being chosen that list was kept the same during different experiments to ensure fairness. That list contains a total of 135 queries, each with 20 seconds of execution time, and begins with a set of all 5 queries being sequentially executed 3 times in every framework regardless our strategy in order to warm up the platform, gather some initial statistics and allow the streaming frameworks to start processing incoming streams smoothly. Following this set of warm-up queries 3 additional sets of queries

have to be assigned to the available at the time frameworks based on a predefined strategy. The 1st set contains 15 queries and each one will be executed once in a framework that will be decided by the strategy at the time of arrival while the rest of the frameworks stay inactive. The 2nd set consists of 45 queries but unlike the 1st set these queries are assigned three at a time which means that every framework will be assigned exactly one query each time. The 3rd and final set consists of 75 queries that will have to be assigned 5 at a time which means that one or more frameworks will be assigned multiple queries at a time. All queries are executed for 20 seconds.

Information about the framework-specific configuration properties:

- Apache Beam (version 2.10.0)
  - --embeddedExecution=false (Apex only)
  - --readTimePercentage=0.2 (Spark only)
  - --batchIntervalMillis=5000 (Spark only)
  - --cacheDisabled=true (Spark only)
- Apache Spark (version 2.3.0)
  - spark.streaming.backpressure.enabled=true
  - spark.scheduler.listenerbus.eventqueue.capacity=100000
  - spark.executor.heartbeatInterval=20
  - spark.memory.fraction=0.6
  - spark.executor.extraJavaOptions=-XX:+UseCompressedOops -XX:+UseG1GC
- Apache Apex (version 3.6.0)
  - apex.application.*.operator.*.attr.TIMEOUT_WINDOW_COUNT=1200
- Apache Flink (version 1.6.0)
  - --filestToStage was explicitly set to the fat jar of Flink Runner.

Windowing configuration options in Beam pipelines:

- Window length: 5 sec.
- Window strategy: Fixed time window.
- Window allowed delay: 0 sec.
- Window trigger: Event Time trigger.

The following tables contains information about the nodes used in the experiments followed by a list of the Kafka topics.

Table 1: YARN containers and Flink resources used in the distributed experiments.

Nodes	CPU Model	CPU cores (per Node)	RAM (per Node)
1 Job Manager	Intel Xeon E5-2430	4	2GB
4 Task Managers	Intel Xeon X3323	4	4GB
8 YARN Containers	Intel Xeon X3323	4	4GB

Table 2: Topics used during experimental evaluation.

Name	Partitions	Description
Spark 1	16	Streamed relation A of Apache Spark (used in JOINs).
Spark 2	16	Streamed relation B of Apache Spark (used in JOINs).
Flink 1	16	Streamed relation A of Apache Flink (used in JOINs).
Flink 2	16	Streamed relation B of Apache Flink (used in JOINs).
Apex 1	16	Streamed relation A of Apache Apex (used in JOINs).
Apex 2	16	Streamed relation B of Apache Apex (used in JOINs).
Data 1	6	Incoming data stream with information about taxis.
Data 2	6	Incoming data stream with information about fares.
Queries	16	Join queries that will be assigned to a framework.
Output	48	Topic used by every framework as a sink.
Latency	1	Beam sink for latency metrics.
Throughput	1	Beam sink for throughput metrics.

5.2 Scalability

In this section we perform experiments with varying degrees of parallelism in the Kafka Streams application and all of the Apache Beam runners to prove the scalability of our modules. The notation $[x, y]$ implies a set of parallelism degree and input stream rate respectively.

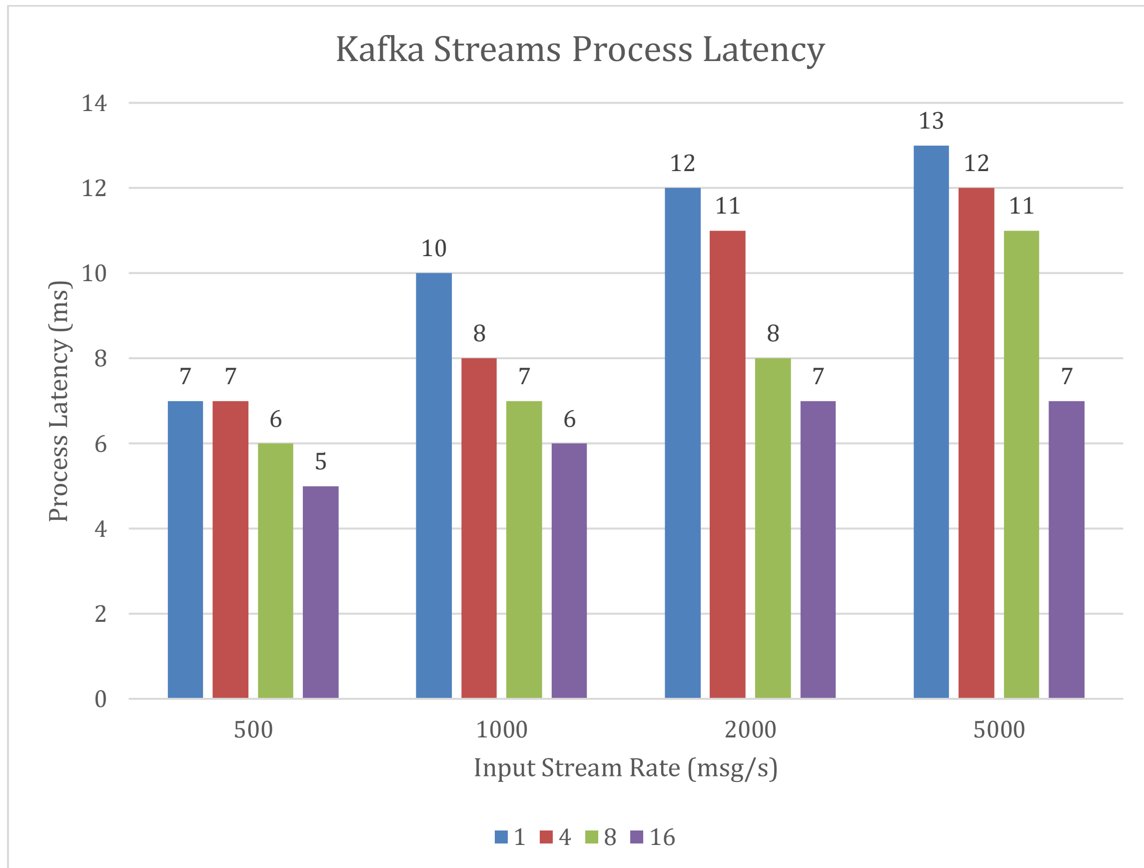

Figure 22: The processing latency of the Kafka Streams application for different number of parallel instances and input stream rates.

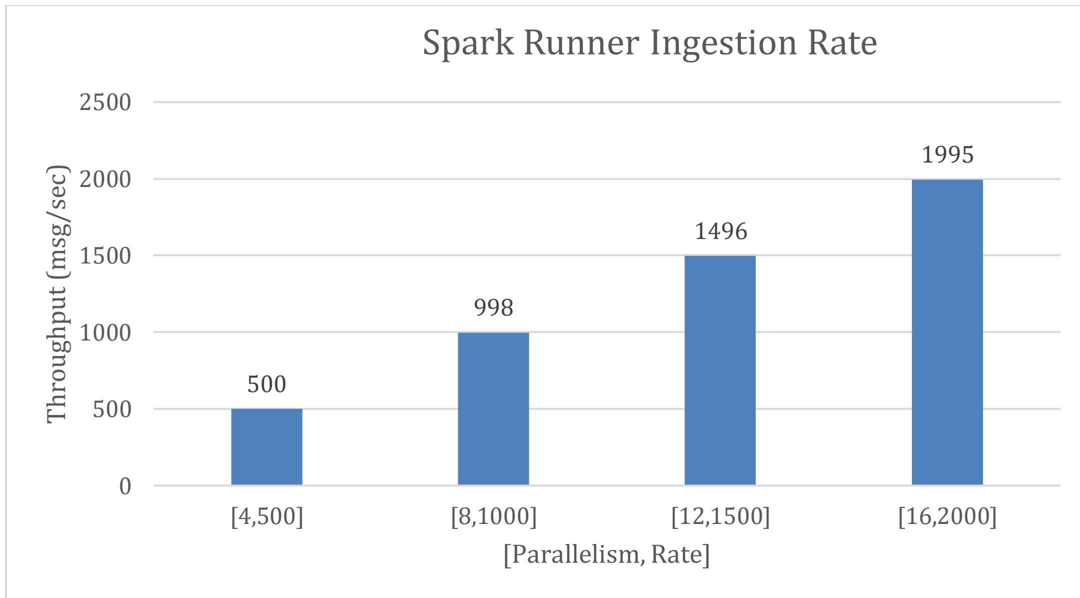

Figure 23: Spark runner throughput for varying parallelism degrees and stream rates.

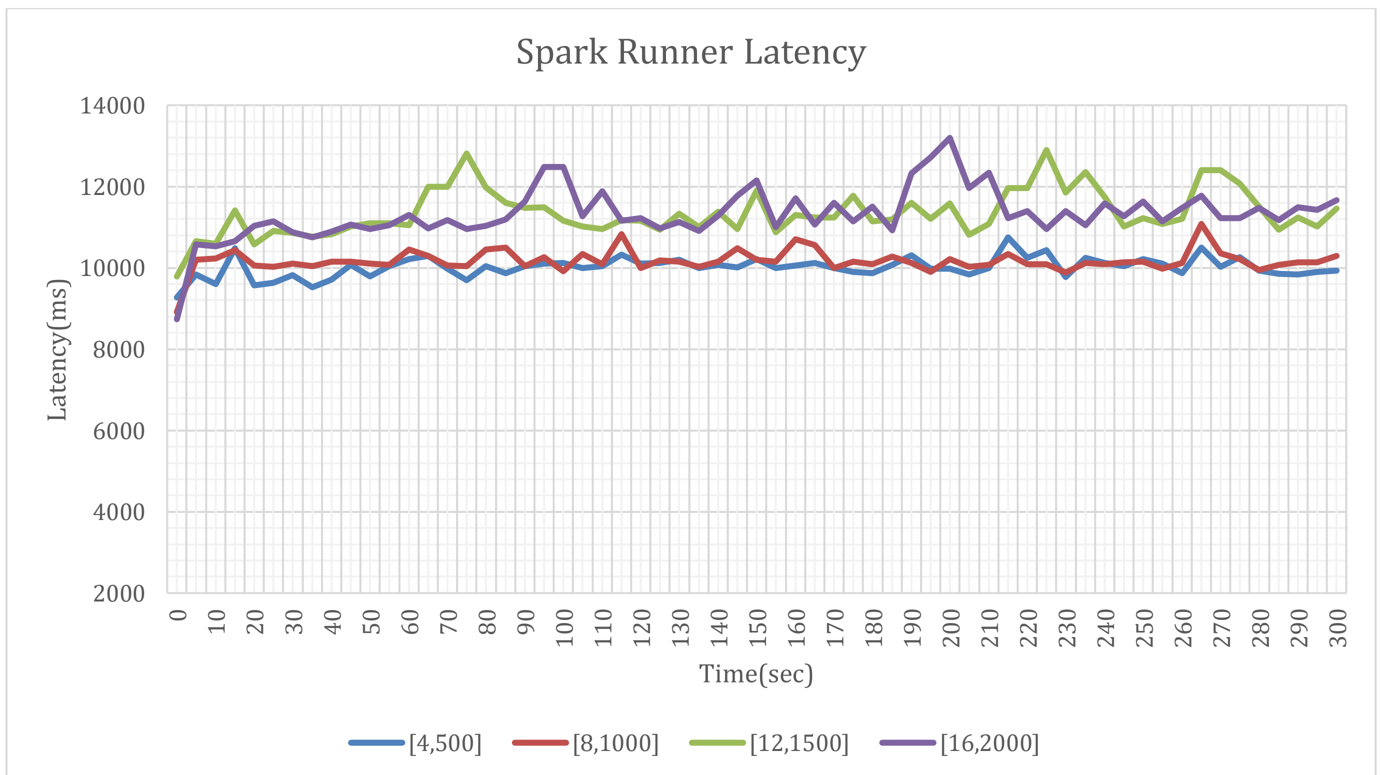

Figure 24: Spark runner latency for different sets of parallelism degrees and stream rates.

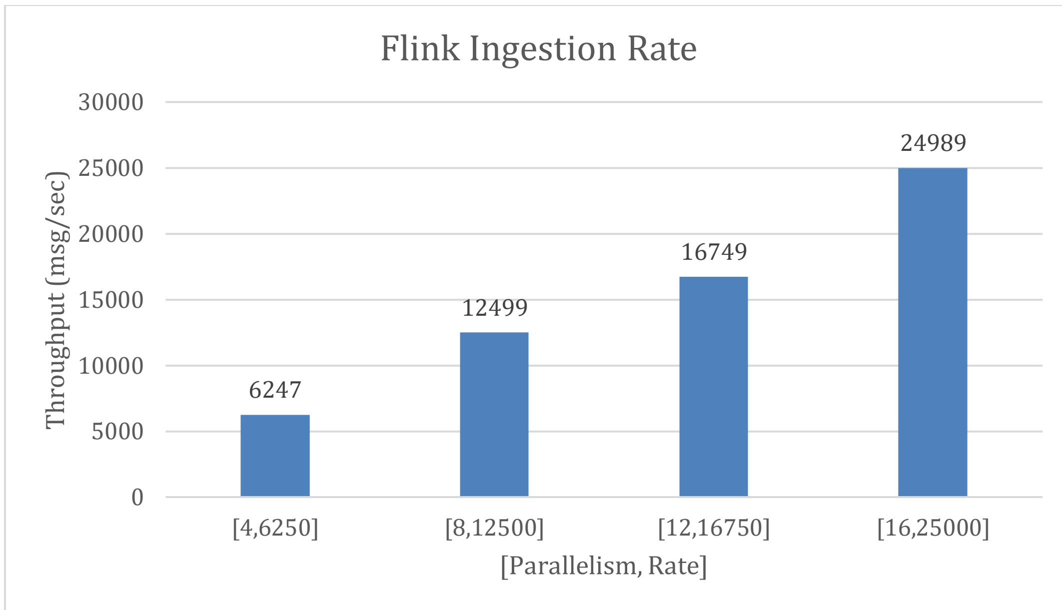

Figure 25: Flink Runner throughput for varying parallelism degrees and stream rates.

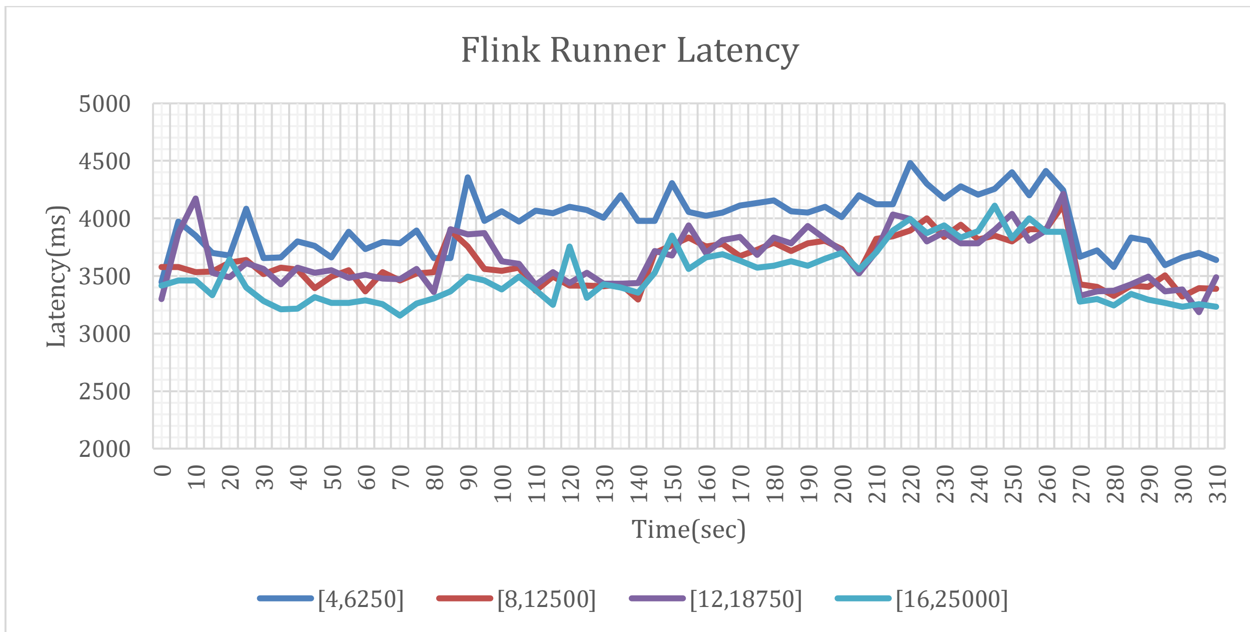

Figure 26: Flink Runner latency for different sets of parallelism degrees and stream rates.

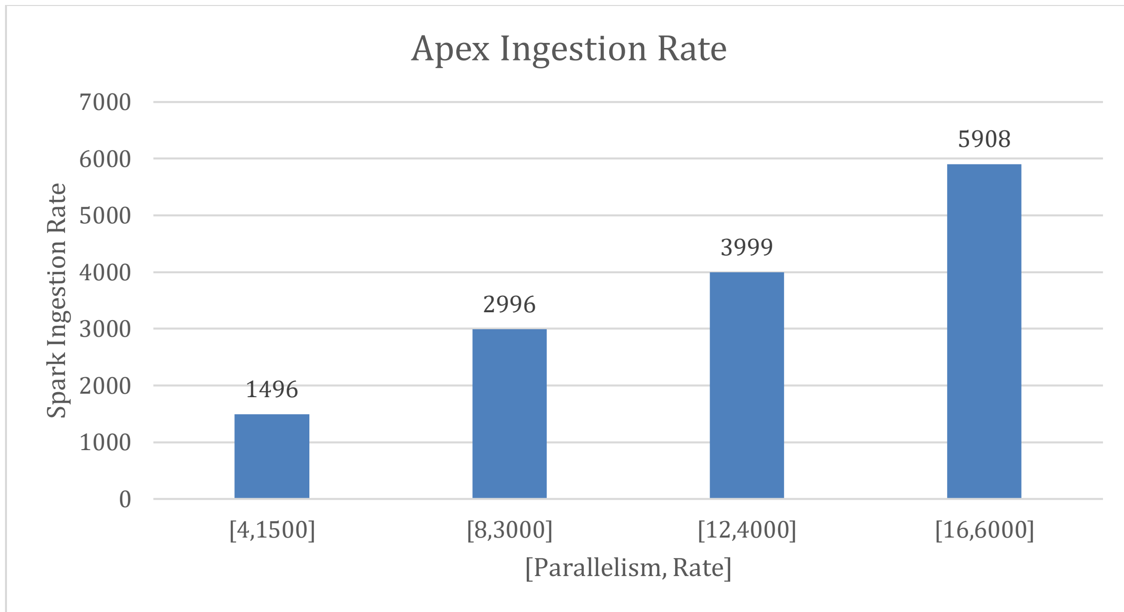

Figure 27: Apex Runner throughput for varying parallelism degrees and stream rates.

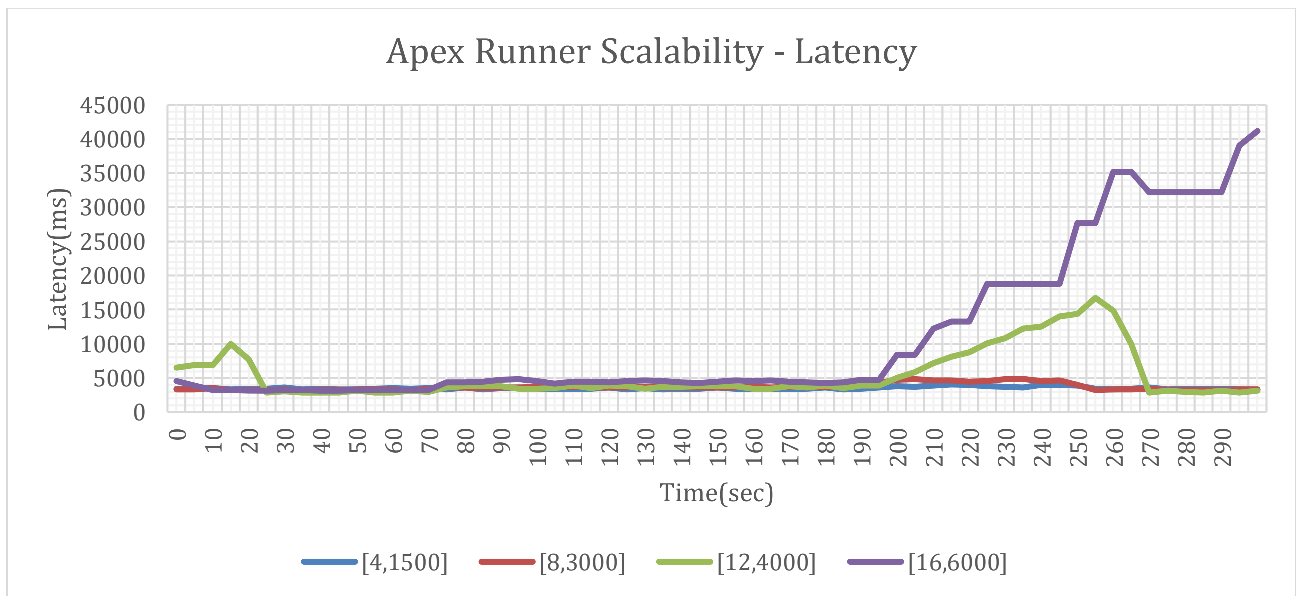

Figure 28: Apex Runner latency for different sets of parallelism degrees and stream rates.

5.3 Strategy comparison

In this section we evaluate the performance of the three strategies, as discussed in the previous chapter, under input streams with different rates. The latency values of each strategy are determined by the framework the current query is assigned on, meaning that a strategy which changes frameworks quickly will also quickly change latency values as well.

The strategies:

- Round robin (RR)
- Flink only (Flink)
- Nearest Neighbor with Lowest Latency (NN-LL or NN)

It's important to note that the latency axes are in logarithmic scale and that the title of each graph has the format {Metric} – {Input stream rate in messages per second}.

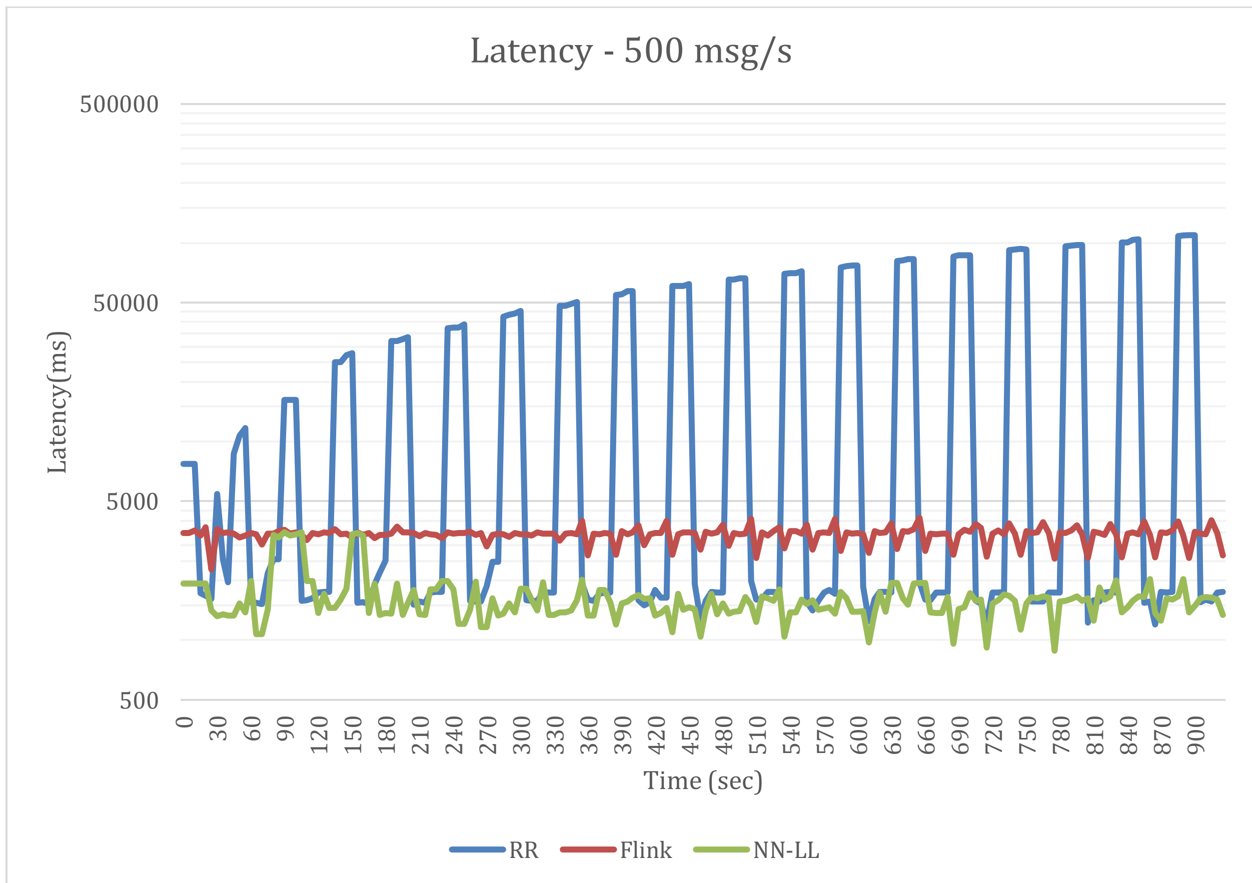

Figure 29: Strategy latencies under input stream rate of 500 msg/s.

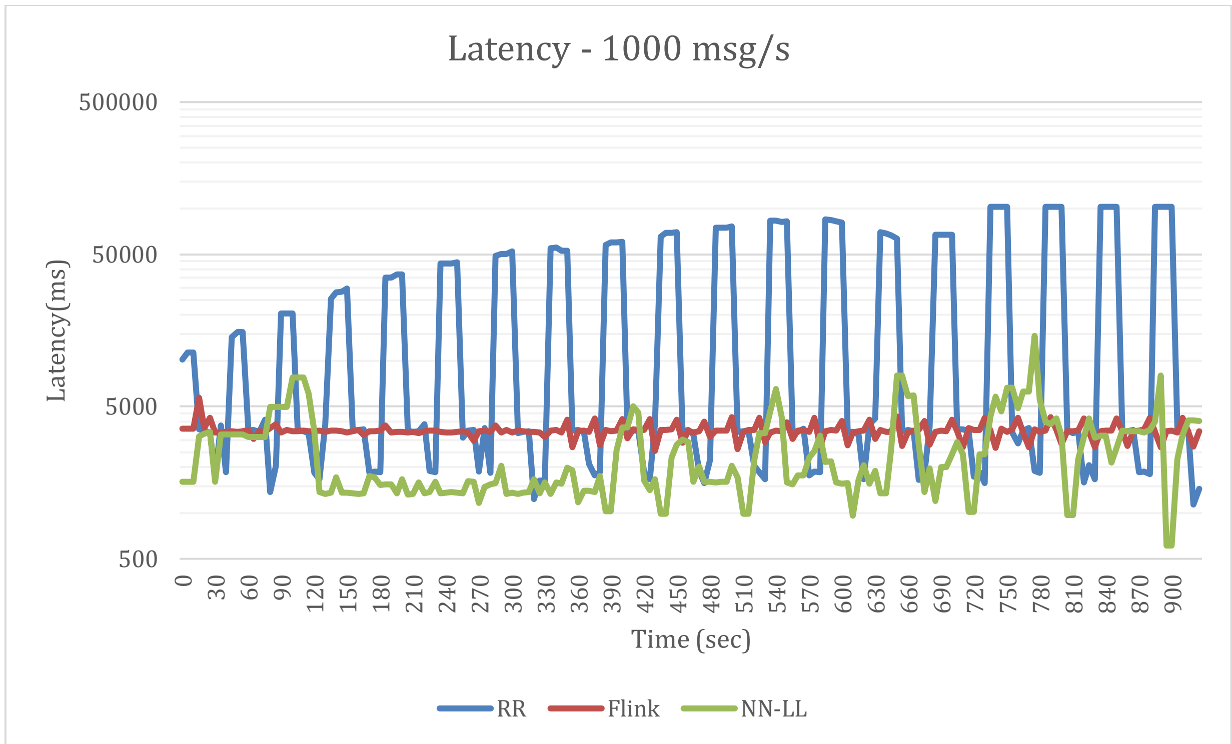

Figure 9: Strategy latencies under input stream rate of 1000 msg/s.

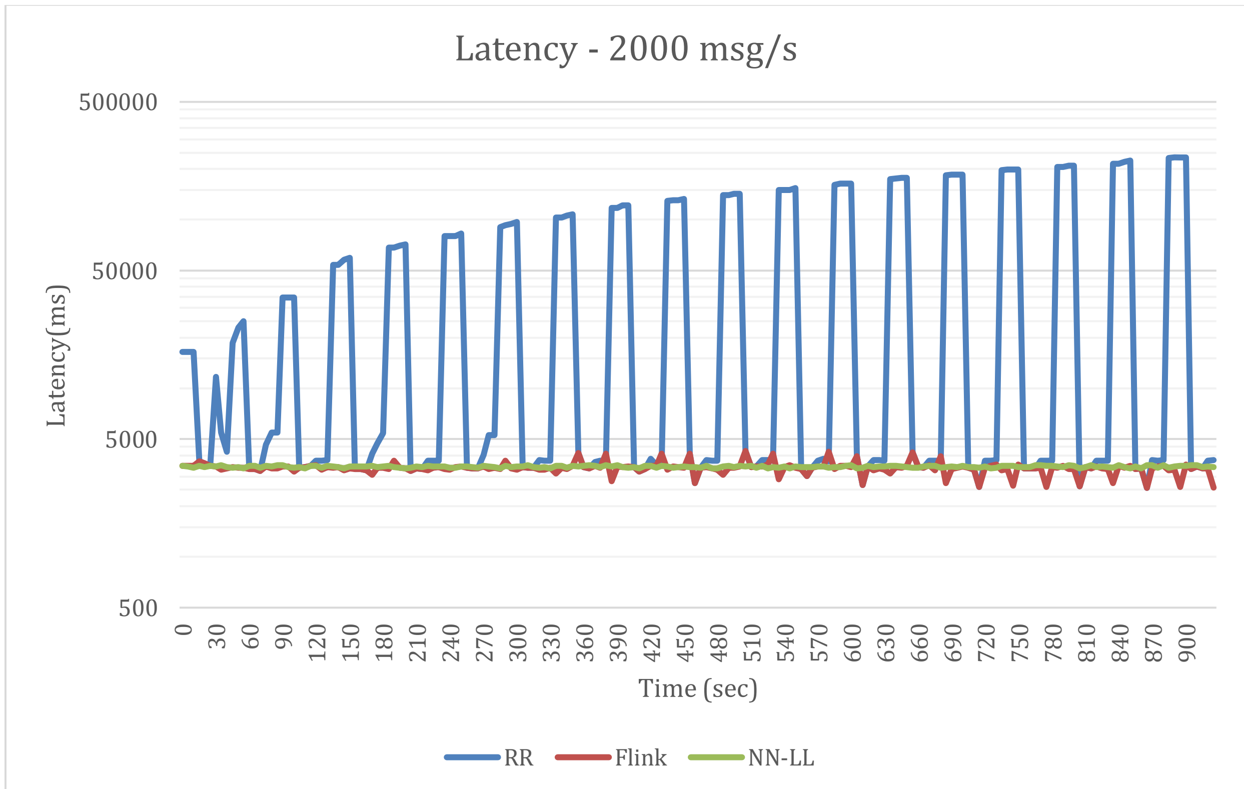

Figure 30: Strategy latencies under input stream rate of 2000 msg/s.

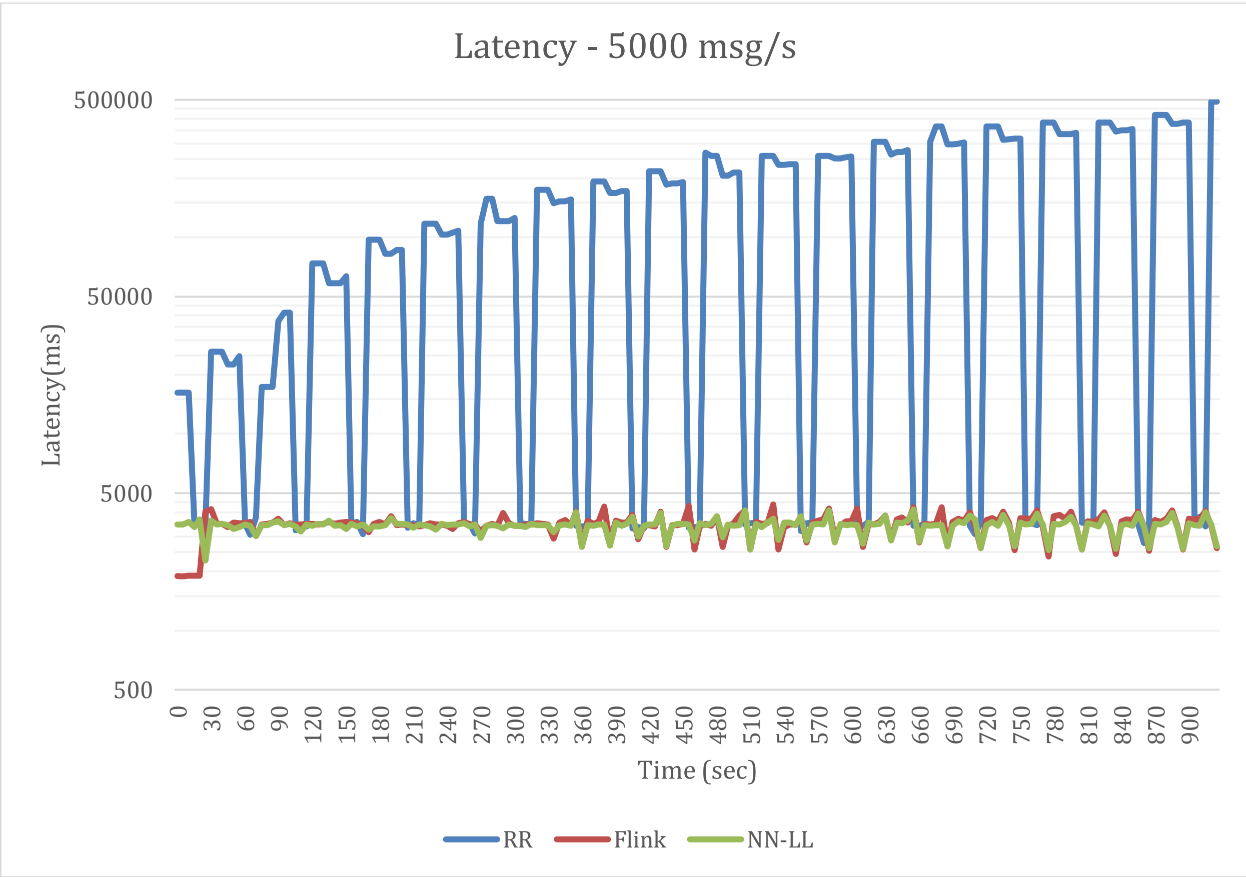

Figure 31: Strategy latencies under input stream rate of 5000 msg/s.

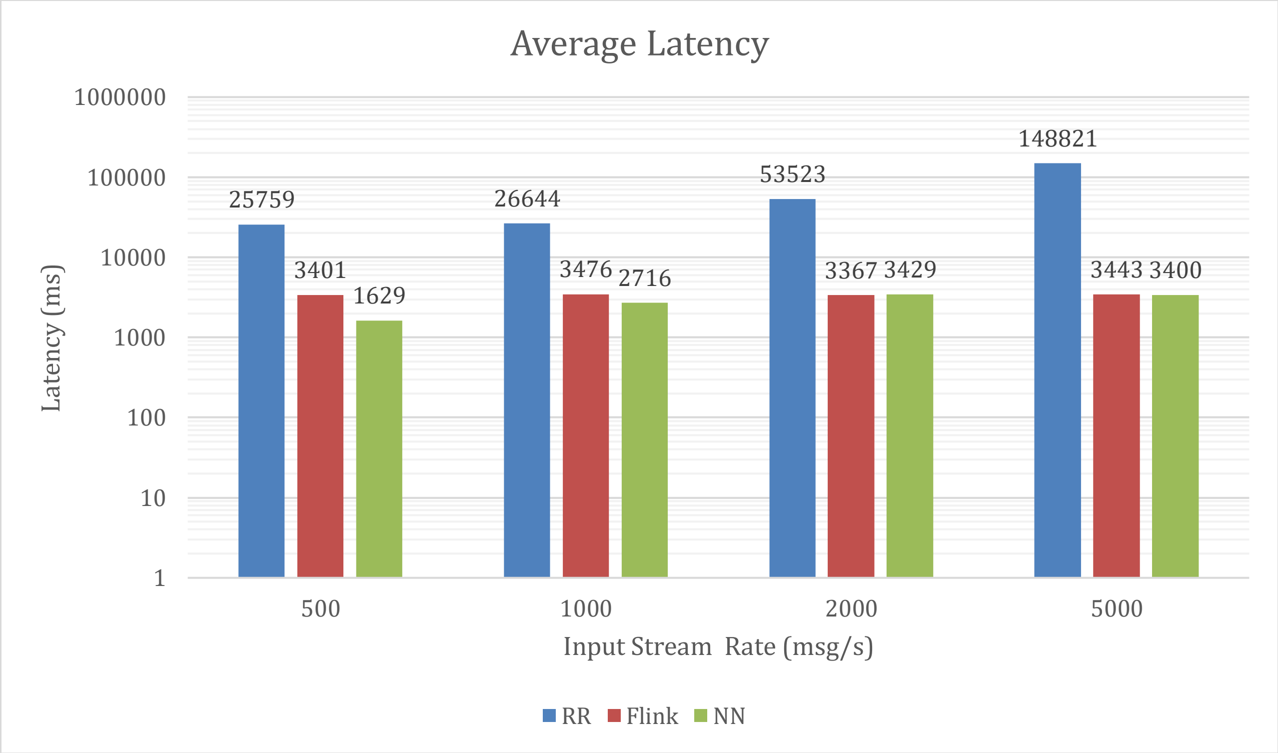

Figure 32: Average latency for each strategy under various stream rates.

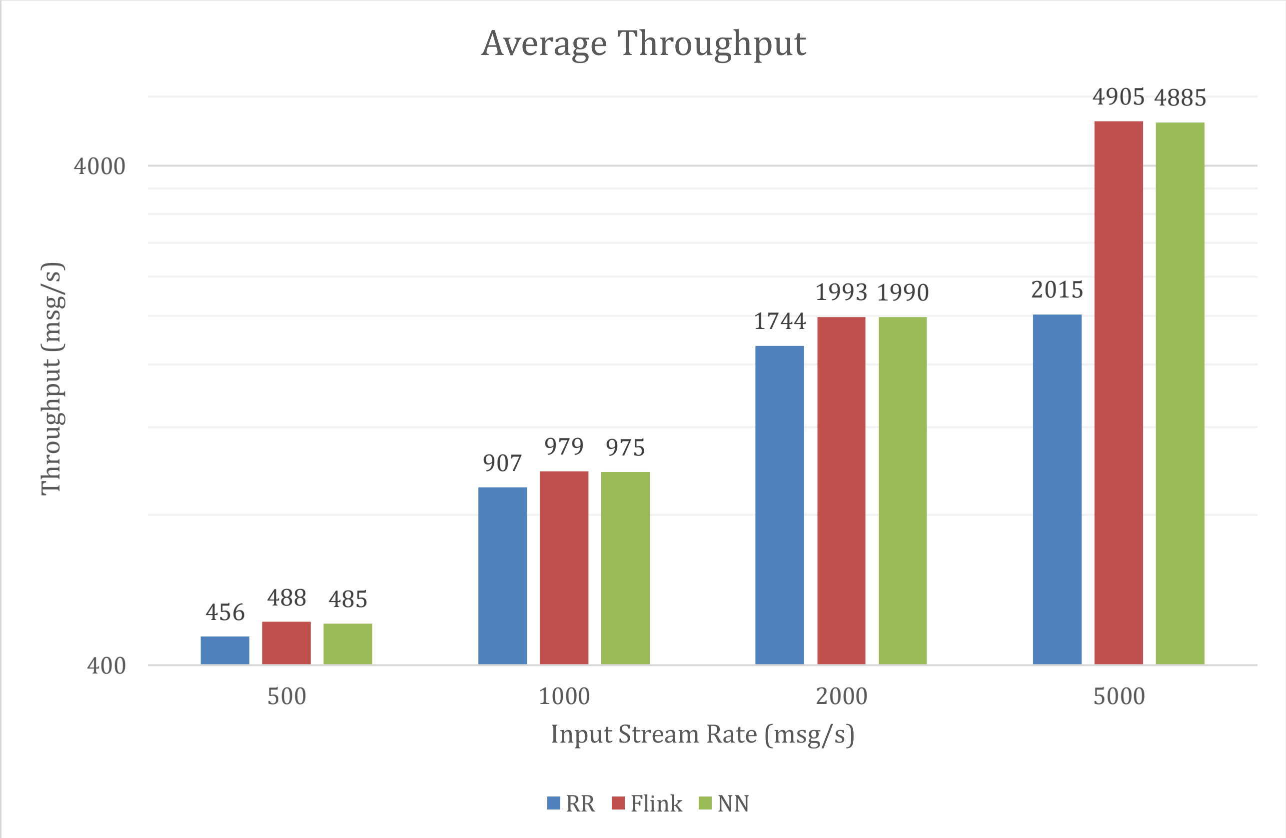

Figure 35: Average throughput for each strategy under various stream rates.

6. Conclusions and Future Work

This thesis proposed a platform that for cross-platform query optimization that assigns queries to various frameworks based on a variety of available metrics and statistics.

The first module of our platform consists of a Kafka Streams application where incoming data streams are sampled before being sent to a framework based on active queries and available statistics. Our Kafka Streams application is scalable, processes tuples at low-latency, has fault-tolerant semantics and can easily aggregate information from its parallel instances. Experiments show that tuple processing latency decreases when more parallel instances are added to the Kafka Streams topology.

Another module of our platform is the Apache Beam dataflow pipeline which can seamlessly be executed on a number of distributed backends. The Beam pipeline contains the implementation of the join queries and will be executed on the backend of our choice. The Apache Beam runners carry out the task of transforming the high-level Beam pipeline to a DAG that will be executed on a distributed environment. However, not every runner performs well, especially when it comes to stream workloads. Experimental evaluation shows that the Spark runner performs poorly when faced with stream workloads since its stream ingestion rate doesn't scale well and its latency although stable is still high. Apache Flink on the other hand performs quite well since the experiments show stable and low-latency under varying input stream rates while maintaining a high stream ingestion rate. Finally, Apache Apex performs better than Apache Flink on input streams with low rate but its performance gradually degrades as the input stream rates rise. However, it's input stream ingestion rate is still on par with Apache Flink's.

The final module of our proposed platform is the Controller, an application that collects available metrics from frameworks, Kafka application instances and Resource managers in order to effectively assign queries to available frameworks. The proposed strategies provide some interesting results. The 1st strategy (round robin) is proven to be the worst one as it underperforms in both latency and throughput under every input stream rate and is therefore not recommended. The main reason behind the weak performance of the first strategy is that it takes no available metrics under consideration and therefore assigns queries to Apache Spark which is proven to be the worst Beam Runner for this set of queries. The 2nd strategy assigns queries only to Apache Flink while the 3rd uses locality sensitive hashing (LSH) to find previous queries that performed well under similar circumstances. Although the 2nd strategy doesn't take into account any of the available metrics it still performs quite well, especially on higher input stream rates. However, the 3rd strategy exhibits lower latency values on lower input stream rates while maintaining almost the same performance with the 2nd strategy on higher input stream rates, therefore the 2nd strategy is recommended.

Although our proposed platform performed well and experiments showed its advantages, it is important to acknowledge that there are things that can be improved in future work. Starting with the number of Apache Beam runners, in the future we could experiment with more frameworks and see if they produce better results. We could also expand the set of queries since currently the number of supported queries is still low. Furthermore, we could also gather more metrics either from stream samples or from framework and resource manager APIs. On that note, the distributed stream sampling algorithm can be improved by using a more modern sampling algorithm with better time and space complexity. Finally, the workload assessment and the

decision-making of the query assignments can be significantly improved by implementing a more sophisticated algorithm and maintaining a different model.

References

[1] "Apache Kafka," [Online]. Available: kafka.apache.org.

[2] "Beam Guide," [Online]. Available: https://beam.apache.org/documentation/programming-guide/.

[3] Zaharia, Matei & Chowdhury, Mosharaf & J. Franklin, Michael & Shenker, Scott & Stoica, Ion. (2010). Spark: Cluster Computing with Working Sets. Proceedings of the 2nd USENIX conference on Hot topics in cloud computing. 10. 10-10.

[4] Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., & Tzoumas, K. (2015). Apache Flink™: Stream and Batch Processing in a Single Engine. IEEE Data Eng. Bull., 38, 28-38.

[5] J. S. VITTER, "Random Sampling with a Reservoir," *ACM Transactions on Mathematical Software, Vol. 11,* pp. 37-57, March 1985.

[6] M. C. Malcolm Slaney, "Locality-Sensitive Hashing for Finding Nearest Neighbors," *IEEE SIGNAL PROCESSING MAGAZINE,* pp. 128-131, March 2008.

[7] T. Akidau, "The Dataflow Model".*Proceedings of the VLDB Endowment, Vol. 8, No. 12.*

[8] Kaggle, [Online]. Available: https://www.kaggle.com/dster/nyc-taxi-fare-bigquery-dataset.

[9] "Apache Beam Documentation," Apache Software Foundation, [Online]. Available: https://beam.apache.org/documentation/.

[10] "Apache Apex Documentation," Apache Software Foundation, [Online]. Available: https://apex.apache.org/docs.html.

[11] "Apache Flink Documentation," Apache Software Foundation, [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-stable/release-notes/flink-1.6.html.

[12] "YARN Documentation," [Online]. Available: https://yarnpkg.com/lang/en/docs/.