Smart-Streams

Vítor Manuel Freitas Escórcio
vitorescorcio@tecnico.ulisboa.pt

INESC-ID, Instituto Superior Técnico, Universidade de Lisboa

Abstract. Nowadays, despite the enormous computing capacity that has become available, large-scale data computing applications continue to exceed available resources. Furthermore, they consume significant amounts of time and energy. A possible approach is to take advantage of approximate computing, where only a representative sample is computed instead of the entire input dataset. Thus, making a trade-off between a result's accuracy and computation speed, and efficiency of resources. We propose a solution, Smart-Streams, which is to implement a smart sampling mechanism on a stream processing system, such as Flink, that will pick the best sampling method to the job at hand, having into consideration factors such as the user requirements and the stream program, while aiming to enforce error bounds.

Keywords: Streaming, Resource Efficiency, Approximation

Table of Contents

1 Introduction

Nowadays, the need for large-scale data computing is increasingly bigger. This is mainly due to the advances in areas such as IOT and Cloud, where data is being created at higher rates than ever before.

In response, solutions are created and despite the enormous computing capacity that has become available, large-scale data computing applications continue to exceed available resources. Moreover, they consume significant amounts of time and energy. There is a lack of solutions with focus on efficiency of computation.

This work is organized as follows: Section 2 details the objectives of this work. Our research on the related work about our proposal is described thoroughly on section 3. Section 4 presents our proposed solution to accomplish the objectives proposed. Section 5 presents how we intend to evaluate our proposed solution. Section 6 concludes and Section 7 presents the work scheduling.

2 Objectives

The goal of this work is to design and develop an extension to a distributed stream processing system, such as Flink, for increased scalability and performance, with adaptable resource management driven by service-level objectives regarding performance (latency, throughput), result accuracy, and resource management.

The core adaptation mechanisms include memory reduction techniques (e.g., compression, summarization) and load shedding (e.g., sampling). The heart of the adaption process will be a hybrid of code annotation and learning-driven process that monitors input tuple distributions and computation results over time.

The combination of these two mechanisms will enable Smart-Streams to determine fits that can allow avoiding storing and/or processing part of the input tuples, thus saving resources and improving performance, while avoiding errors to exceed application-defined bounds.

3 Related Work

3.1 Stream Processing

Up until recently, the vast majority of data being processed was done via batch processing. Batch processing is the processing of data in batches i.e. bounded (finite) data is being processed by a system in batches. However, stream processing has been gaining increasing popularity in the past few years, and for good reasons. Stream processing is a type of data processing engine that is designed with infinite data sets in mind (this include micro-batch i.e., batch with few events and true streaming i.e., tuple at a time). Some of the reasons include:

- Businesses crave ever more timely data, and switching to streaming is a good way to achieve lower latency.
- The massive, unbounded data sets that are increasingly common in modern business are more easily tamed using a system designed for such never-ending volumes of data.

– Processing data as they arrive spreads workloads out more evenly over time, yielding more consistent and predictable consumption of resources.

Despite this business-driven surge of interest in streaming, the majority of streaming systems in existence remain relatively immature compared to their batch counter-parts, which has resulted in a lot of active development in the area recently.

3.1.1 Typical Use Cases. There are quite a few applications that cannot be realized with batch processing technology due to their demanding latency requirements but which are a perfect fit for modern stream processing. Such applications include:

- Real-time recommendations [1]: To make recommendations in real time requires the ability to correlate product, customer, supplier, logistics and even social sentiment data. It also requires the ability to instantly capture any new interests shown in the customers' current session visit. Something that batch processing systems cannot accomplish.
- Pattern detection or complex event processing (CEP) [2]: CEP identifies meaningful events such as opportunities or threats and responds to them as quickly as possible. It does this by matching continuously incoming events against observed patterns. CEP is particularly useful for cases such as fraud detection [3] in credit card transactions, detecting network intrusion by specifying patterns of suspicious user behavior, and financial applications, for instance, stock market trend;
- Online ETL [4]: to continuously extract, transform and load data as it is produced into data warehouses so, for instance, analytics end users have faster access to data;
- Sentiment Analysis [5]: also known as opinion mining, aims to determine the attitude or opinion of a subject with respect to a topic or something. With text analysis of reviews and surveys responses, it answers practical business questions such as "Why aren't consumers buying our laptop?" and allow them to quickly react.

3.1.2 Architecture evolution. Streaming systems have long been relegated to a somewhat niche market of providing low-latency, inaccurate/speculative results, often in conjunction with a more capable batch system to provide eventually correct results, i.e. the **Lambda Architecture**.

The basic idea is that, as shown in Figure 1, you run a streaming system alongside a batch system, both performing essentially the same calculation. The streaming system, e.g. Storm[7], gives you low-latency, to some extent inaccurate results (either because of the use of an approximation algorithm, or because the streaming system itself does not provide correctness), and some time later a batch system e.g. Hadoop [8], rolls along and provides you with correct output. In the beginning, streaming engines were a bit of a letdown in what correctness

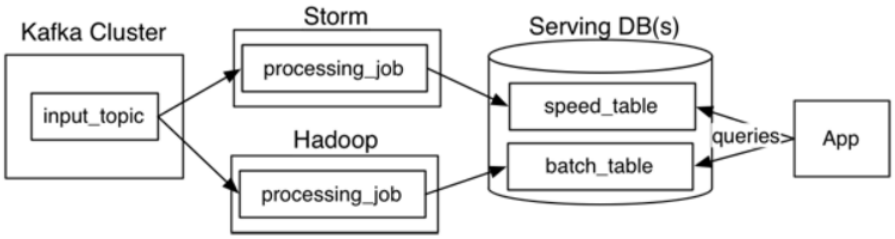

Fig. 1. Lambda Architecture [6]

is concerned, and batch engines were as inherently slow regarding latency as it is expected. That was the main motivation to join them.

Unfortunately, maintaining a Lambda system is very costly [6]: you need to build, provision, and maintain two independent versions of your pipeline, and then also somehow merge the results from the two pipelines correctly at the end, that may be non-trivial.

For a while, the Lambda architecture was enough. However, because of its limitations and overheads, and with the growing requirements of low-latency with correct results, the Lambda architecture became obsolete. With the maturation of stream processing systems and evolving frameworks to deal with unbounded data (e.g. Kafka [9]), a new architecture emerged.

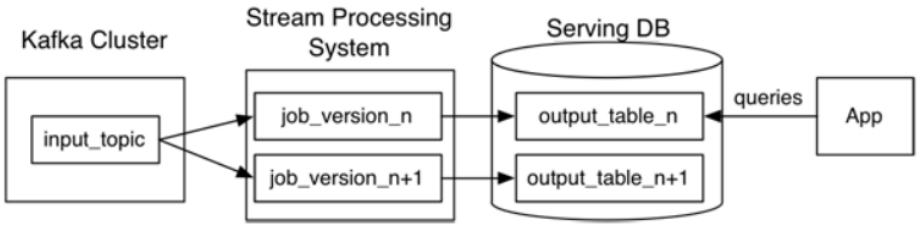

Fig. 2. Kappa Architecture [6]

Kappa Architecture. The idea is, as illustrated in Figure 2, to run a single pipeline using a well-designed system that is appropriately built for the job at hand. It handles both real-time data processing and continuous reprocessing in a single stream processing engine. This requires that the incoming data stream can be replayed, either in its entirety or from a specific position. If there are any code changes, then a second stream process would replay all previous data through the latest real-time engine and replace the data stored in the serving layer.

3.1.3 Programming Model can be classified as **Compositional** or **Declarative**. Compositional approach provides basic building blocks like sources or operators and they must be linked together in order to create an expected topology. New components can be usually defined by implementing some kind of interfaces. On the contrary, operators in declarative API are defined as higher order functions. It allows to write functional code with abstract types and the system creates and optimizes topology itself. Also, declarative APIs usually provides more advanced operations like windowing or state management more easily.

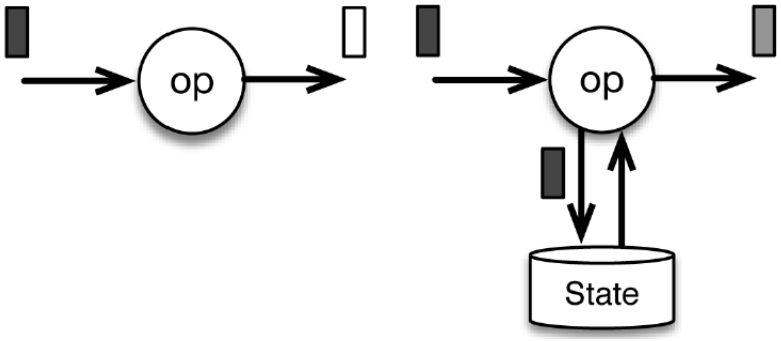

Fig. 3. Stateless vs stateful operation

3.1.4 Operations on data streams. Nowadays, stream processing systems provide users with a set of built-in operators to ingest, transform and output events or streams of events. These operations can be either stateless or stateful operations. Stateless operations do not maintain any state after processing an event. In other words, transformations of events are independent from one another. This allows for easy parallelization of operations, even when in presence of unordered events. In contrast, stateful operations are operations in which some sort of state is maintained over time. This state can then be updated by incoming events which is used for the processing logic. Stateful operations are harder to parallelize because the state needs to be efficiently partitioned and recovered reliably in case of failure.

Data ingestion and output. Data ingestion and output operations allow stream processing systems to better communicate with external systems. Data ingestion operations serve the purpose of fetching raw data from external sources and transform them into a more suitable format to be processed by other operations. Operators with this purpose are usually named as Sources. Data output operations transform processed data into suitable data formats so they can then serve as input for external systems such as Kafka. These operators are usually named as Sinks.

Transformation operations. Transformation operations are operations that process each event once and independently. These operations essentially process events one at a time and apply a transformation to the event and outputs its result to the next operator. These transformations can be integrated within the operator it self, or can be defined by a programmer.

Operators can receive events from multiple streams and also output multiple streams.

Aggregation operations. Aggregation operations are aggregations, such as a sum, max and min, such that each incoming event updates the operation's state. Thus, aggregation operations are stateful. Note that for these type of operations, it is of paramount importance that the update functions of the state must be very efficient and somewhat associative and commutative, otherwise a complete record of all events would be necessary, which would lead to a big memory slowdown problem.

Window operations. Operations seen so far process a single event at a time. However, some operations require the collection of events to compute their result. An example of these operations would be holistic aggregate operations such as a median. On a stream processing system where the input is constituted by unbounded data, it is unfeasible to keep a record of all data to then compute its result, so a mechanism to limit the data maintained for these operations is needed. This mechanism is called Windowing.

Apart from its practical value, windows also provide interesting stream query semantics. It allows to query the most recent data from the stream. This serves many practical usages. For instance, in an application that provides real-time traffic information to drivers so they can avoid congestion, they only need to know if an accident happened in certain location in the last minutes or hours, not in the last week or if ever occurred an accident in the location. Furthermore, by reducing the stream results just to an aggregate, we lose information about the variation of the data in the stream so, for instance, it might be helpful to know how many vehicles crossed a road every 5 minutes.

Windowing essentially slices a dataset into finite chunks for processing as a group. Windowing is effectively time based. While many systems support tuple-based windowing, in essence, this is time-based windowing over a logical time domain where elements in order have successively increasing logical timestamps. Windows can be aligned, i.e, applied to all data in the window or unaligned, i.e., only applied to a subset of the dataset in the window, e.g., per key. Figure 4 highlights three of the major types of windows encountered when dealing with unbounded data.

Fixed windows. Fixed Windows, also known as Tumbling windows, are defined by a fixed window size, i.e., temporal length. Usually, as in a) in Figure 4, the segments of fixed window size are applied in the same manner across all the dataset. Then, when window borders are passed, all data (in case of aligned) or some data (in case of unaligned) in the window is evaluated by a function. In

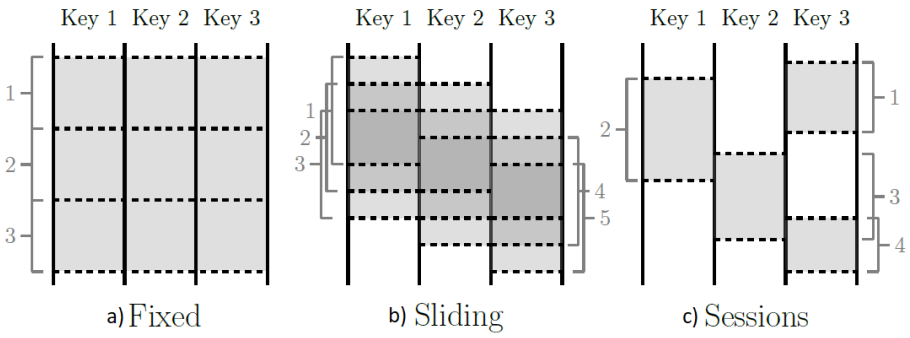

Fig. 4. Common Windowing Types [10]

count-based fixed windows, the size of the window is defined through a number of events instead of a temporal length.

Sliding windows. Sliding windows are a generalization of fixed windows where the size and slide period are fixed. If the period is less than the size, the windows overlap, as in b) in Figure 4 (most frequent case). If the period equals the size, you get fixed windows. And if the period is greater than the size, you get a sort of sampling window that does not look into the data between the windows. The main goal is to overlap the windows in a way that the most recent data results one has, e.g., windows with an hour size and with a minute of sliding period. This way, every minute one has the results of the data seen in the last hour.

Session windows. Session windows are dynamic, that is, they do not have fixed size. Instead, size is dynamically defined by some period of activity terminated by a timeout of inactivity. Any events that occur within a span of time less than the timeout are grouped together as a session. Session windows are commonly used to analyze user behavior over time, by grouping together a series of temporally related events. In this case, as in c) in Figure 4, per key. While fixed and sliding windows are for the most part aligned, sessions are unaligned.

3.1.5 Time semantics. When dealing with unbounded stream of continuous incoming events, time becomes a central aspect of applications. In real world systems and applications, unfortunately, the events may arrive out-of-order and with varying event-time skew (illustrated in Figure 5), which might influence the program logic and results. This is due to a variety of reasons. Some sources of disorderliness introduced are the following:

- Producers of the events have clock skews. This is common when producers are from different machines, so they have different clocks.
- Shared resource limitations, such as network congestion, network partitions, or shared CPU in a non-dedicated environment.

- Software causes, such as distributed system logic, contention, etc.
- Features of the data themselves, including key distribution, variance in throughput, or variance in disorder (e.g., a plane full of people taking their phones out of airplane mode after having used them offline for the entire flight).

Within any data processing system, there are typically two time domains considered in which stream processing systems operate by.

Processing time. Processing time is the time at which an event is observed at any given point during processing within the pipeline. A processing-time window includes all events that happen to have arrived at the window operator within a time period, as measured by the local clock of the machine where the operator processing the stream of events is being executed.

Event time. Event time is the time at which an event actually occurred. Event time is based on a timestamp that is attached to the events in the stream. These timestamps usually exist inside the event prior to its ingestion in the pipeline e.g. created as a record of the system's clock of origin, when the event was created.

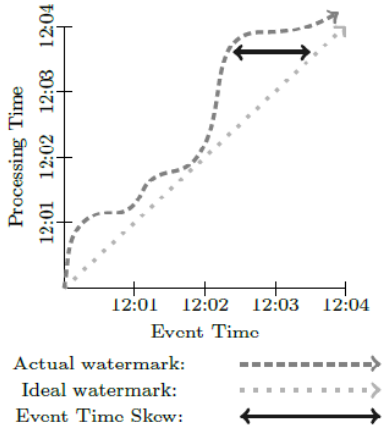

Fig. 5. Time Domain Skew

Processing time vs Event time. Event time completely decouples the processing speed of events from the results. Operations based on event time are predictable and their result are deterministic. In other words, computing a stream of events should always end with the same result, regardless of the processing time and disorderliness of arriving events.

Handling out of order events, due to causes mentioned previously, is one of the big advantages of processing based on event time. While event time guarantees

result correctness even with out of order events, there is an overhead of waiting by late events and ordering events, as we will see further ahead with watermarks.

In contrast, processing time windows introduces the lowest latency possible. Since it does not consider out of order events, the window simply buffers events and triggers its computation as soon as the temporal length has passed. Thus, processing time is most suitable for applications where low latency is more important than accuracy.

Watermarks. So far, we have seen that event time guarantees deterministic results and allows us to deal with events that are late and/or out-of-order, but not how. Given the unpredictable reality of distributed systems and arbitrary delays that might be caused by external components, it is hard to know how long one has to wait to be certain that one has received all data before triggering a computation or even know if data will be delayed in the first place.

Watermark is a notion of input completeness with respect to event times i.e. it acts as a metric of progress when observing an unbounded data source with no known end . In essence, watermarks provide a logical clock that informs the system about the current event time. When a watermark is received with a value of X, the operator can assume that no further events will be received with a time stamp inferior to X. Watermarks are essential to both event time windows and operators that deal with out of order events. When a watermark is received by an operator, these are signaled that all events with timestamp inferior to the watermark have been received and it trigger computation or order events.

Watermarks provide a configurable trade-off between accuracy of results and latency. In many real world applications, the system does not have enough information to perfectly create watermarks. Whether watermarks are user-defined or automatically created watermarks, tracking global progress in a distributed system is hard and problematic, specially in the presence of stragglers tasks [1]. Therefore, relying only on watermarks might not be a good idea. Systems should provide some mechanisms to deal with events arriving after the watermark. Some possible solutions, depending on their application requirements, could be to ignore those events, log them, or use them to correct previous results.

As we have seen with stateful operations, storing state is of paramount importance in big data processing systems [11]. Not only to have stateful operators, but also to achieve fault tolerance, state must be kept in a reliable location, to enable efficient recovery. State can facilitate the iterative and incremental processing of most machine learning algorithms, where iteration is inevitable. State can also help in achieving load balancing and elasticity i.e., a system capability to scale out when presented with dynamic changing queries or event workload.

3.1.6 Processing Semantic Guarantees. Stream processing systems provides guarantees with regard to processing of events and consistency of its internal state, upon task failures. We can divide these into three different processing

[1] Straggler tasks or Stragglers are tasks within a stage that take much longer to execute than other tasks

semantic guarantees:

At-most-once. The simplest thing to do when a task failure occurs is to do nothing to recover lost state and/or replay lost events. This is the easiest and most performant guarantee because it can be achieved in a fire-and-forget fashion without the need of keeping state. However, this implies that there is no mechanism to ensure result correctness. Although having no guarantee of result correctness sounds like a bad idea, it might work just fine for applications where the minimum latency possible is required and approximate results are enough.

At least once. In most real-world applications, the minimum requirement is that events are processed at least once. That is, contrary to at-most-once guarantee, events do not get lost, even though some events might get duplicated. This guarantee might be acceptable for applications where the correctness of the application only depends on the completeness of information. For example, to determine if a specific event occurs in the input stream, this guarantee is enough. However, if you need to count (or otherwise take into account) how many times that specific event occurs in the input system, this guarantee isn't enough since you would probably end up with a higher count due to duplicates.

To ensure at-least-once, there is a need for a mechanism that can replay events, either from the event source or from some buffer. Persistent event logs write all events into durable storage so they can be replayed later on if needed. Another popular approach is with record acknowledgement. In this approach, events are stored in a buffer and discarded after the processing of the events are acknowledged. If not, they are resent.

Exactly once. This is the strictest and most challenging to achieve type of guarantee. Exactly-once result guarantee means that not only there will be no event loss, but also events will be processed no more than once, even if there are duplicates. In essence, the result will be correct as if a failure never occurred.

Providing exactly-once guarantees require at-least-once guarantees, therefore a replay event mechanism is again necessary. Additionally, the system needs to have a duplicative detection mechanism. That is, the system needs to know if an event has already been processed and discard its duplicates, after it has been processed once. Another popular approach is through distributed snapshots/state checkpointing where all states in the application is periodically checkpointed and in case of a failure in the system, states are brought back to the most recent globally consistent checkpoint/snapshot.

3.1.7 Dataflow graphs A dataflow program, as the name suggests, describes how data flows between operations. Dataflow programs are commonly represented as directed graphs, where nodes are called operators and represent computations and edges represent data dependencies. Operators are the basic functional units of a dataflow application. They consume data from inputs, perform a computation on them, and produce data to outputs for further processing.

Operators without input ports are called data sources and operators without output ports are called data sinks. A dataflow graph must have at least one data source and one data sink, as pictured in Figure 6.

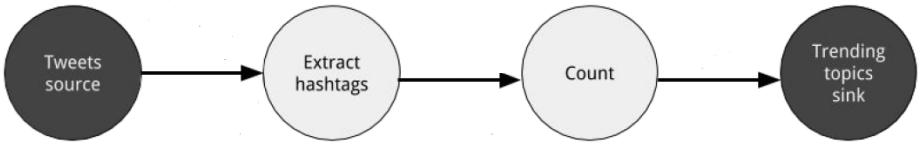

Fig. 6. A logical dataflow graph to continuously count hashtags

Dataflow graphs like the one of Figure 6 are called logical because they convey a high-level view of the computation logic. In order to execute a dataflow program, its logical graph is converted into a physical dataflow graph, which includes details about how the computation is going to be executed. For instance, if we are using a distributed processing engine, each operator might have several parallel tasks for each operator. Figure 7 shows a physical dataflow graph for the logical graph of Figure 6. While in the logical dataflow graph the nodes represent operators, in the physical dataflow, the nodes are tasks. The Extract hashtags and Count operators have two parallel operator tasks, each performing a computation on a subset of the input data.

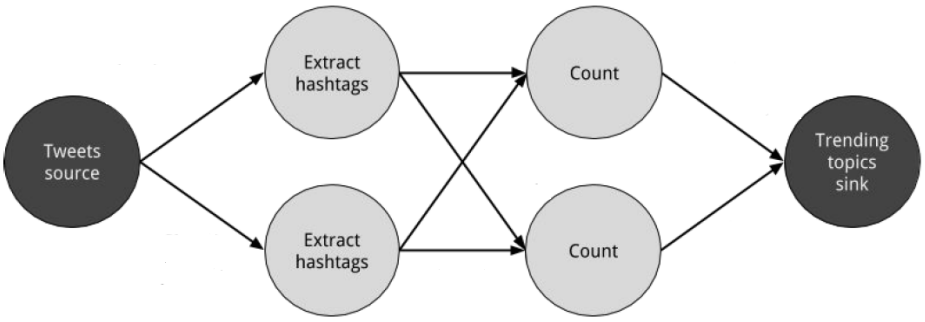

Fig. 7. A physical dataflow plan for counting hashtags

Physical distribution. Two types of physical distribution of pipelines in stream processing systems can be seen in practice. In the first type there is a central server that orchestrates the distribution of the operators into physical nodes. This node is responsible for load balancing the tasks amongst the nodes and also schedule the tasks to run on different nodes. In the other type of processing engines there is no central server to orchestrate the work among the nodes, and the nodes work as a fully distributed peer-to-peer system sharing the work

equally. In both models a single operator can be running in parallel on different nodes. Usually, pipelines are sliced in a way that senders and receivers with data dependency remain in the same machines to avoid network communication.

Data routing. Data exchange strategies define how data items are assigned to tasks in a physical dataflow graph. Data routing strategies can be automatically chosen by the execution engine depending on the semantics of the operators or explicitly imposed by the dataflow programmer. Here, we briefly review some common data routing strategies.

- **Forward** strategy sends data from a task to a receiving task. If both tasks are located on the same physical machine (which is often ensured by task schedulers), this exchange strategy avoids network communication.
- **Broadcast** strategy sends every data item to all parallel tasks of an operator. Because this strategy replicates data and involves network communication, it is fairly expensive.
- **Key-based** strategy partitions data by a key attribute and guarantees that data items having the same key will be processed by the same task. In Figure 7, the output of the Extract hashtags operator is partitioned by key (the hashtag), so that the count operator tasks can correctly compute the occurrences of each hashtag.
- **Random** strategy uniformly distributes data items to operator tasks in order to evenly distribute the load across computing tasks.

3.1.8 Fault Tolerance. Fault tolerance, often due to strict requirements in distributed systems, is very hard to implement. More specifically, stream processing systems [12] differ from other distributed systems such as batch processing systems, databases, etc, in the requirement of real time data processing.

Stream processing systems operate on unbounded streams, rather than bounded data such as a file. These systems can not easily predict trends for incoming data and require low-latency processing of data. Thus, they tend to apply lightweight operations like filters, aggregations, rather than heavy input/output operations. Due to the requirements mentioned, fault tolerance techniques used in other distributed systems cannot be directly applied in stream processing systems. Stream processing systems require fault tolerant techniques with very low recovery time to support time-critical, real-time and continuous applications.

FAULT TOLERANCE APPROACHES

Fault tolerance approaches for stream processing systems can be categorized into three categories. Few solutions use combination of these approaches.

Active Replication Active replication uses operator replication and process each client node on multiple operator instances. It provides a low recovery time at the cost replication, which is not very cost effective for most systems. Active replication involves two expensive steps that are ordered multicast and output synchronization. The first one ensures that all nodes receive the input in

the same order. Whereas the second one ensures delivery of correct and single output to the client. Most of the information processing using active replication are redundant and require at least twice more storage and processing resources. Thus, it is an expensive fault tolerance mechanism

Passive Replication Also known as rollback recovery, is a replication mechanism in which nodes are arranged in master-slave relationship. Only one front-end (the master) server handles the client request. Other nodes in the cluster act as a backup nodes, which are responsible for storing snapshots/checkpoints of the master node. Backup nodes take over the primary node in case of failure and recover from the stored checkpoints. It is more resource efficient but incurs a periodic overhead due to state synchronization. Commonly, passive replication is used along with upstream backup for precise recovery of an operator, i.e., keep the tuples after the checkpoint in the output buffer and use them during recovery phase.

Upstream Backup Upstream backup requires nodes to buffer events until they have been processed by downstream operators, which will acknowledge the events after processed. This scheme is different than active and passive replication in the way it handles failures. Upstream backup uses the output queues at different workers as a temporary storage to store information. This scheme is complicated as multiple downstream operators process a single tuple. The upstream backup model is feasible for operators whose internal state depends on a small amount of input. It is often used in combination with passive replication. In case of failure, the upstream primaries replay their logs and the secondary nodes rebuild the missing state. When used along with passive replication, nodes can recover its state using the last checkpoints and the upstream tuples with timestamp after the last checkpoint.

3.1.9 Performance.

For batch applications, we usually care about total execution time of a job as a whole. Since streaming applications run continuously with potentially infinite, unbounded streams of data, there is no real notion of total execution time. Instead, stream applications focus on delivering the result of incoming data as fast as possible while being able to consume a high rate of events. We express these performance requirements in terms of throughput and latency.

Latency indicates how long it takes for an event to be processed. In other words, it is the time frame between consuming an event and observing its effect in the output.

In stream processing, latency is measured in units of time, for instance, in milliseconds. Depending on the application goal, we care about the overall average latency. However, average latency is not always enough to be ensured because it hides the true distribution of processing delays. This can result in cases where we have an average latency of, for instance, 20ms with peaks of 10s which, for

some applications such as fraud detection, raising alarms and services with strict SLAs with regard to latency, might just not be tolerable. So, in some cases, it might be more helpful referring to percentile (e.g. 95%) or maximum latency.

Modern stream processors can achieve latencies as low as a few milliseconds. In contrast, in batch processors, latencies typically ranges from few minutes to various hours. This happens because, in batch processing, the data is gathered first and only then it can be processed. As a result, the latency is dependent on the last data that will be gathered in the batch before being processed. In true streaming model, this added latency does not exist and the data is processed as soon as it arrives, making latency only dependent on the work needed to be done for each event. This allows for really low latencies and is what makes stream processors so attractive.

Throughput is the rate at which the system can process events ,i.e., number of events per time unit. While we want the latency to be as small as possible, we generally want the throughput to be as high as possible.

Throughput is usually measured in events per time unit, for instance, 120 millions of events per second. It is important to mention that the rate of processing also depends on the rate of arrival and that low throughput not always represents low performance. The system only needs to be able to process events with a throughput as high as the rate of arrival. When the resources of a system are completely unused, as the first event arrives, it will be processed as fast as possible, achieving the lowest latency. It is desirable that this latency remains constant however, as the event arrival rate increases, the throughput also increases, and when it gets to the point where all resources are being used, we reach the peak throughput. Further increasing the event arrival rate, messages start to get buffered and latency starts increasing as well. Further increasing the event arrival rate, it can happen that event buffers become full and events can be lost. This phenomenon of a system receiving data at a higher rate than it can process is known as backpressure and there are some techniques to deal with it [13] .

Latency vs. Throughput. At this point it should be clear that throughput and latency are somewhat related. If events take long traveling in the pipeline, it is not as easy to guarantee throughput. In a similar way, if the peak throughput of a system is relatively low, events are buffered and await before getting processed.

Two ways we could improve performance is with vertical and horizontal scaling. With vertical scaling, latency is immediately improved, and because events are processed faster in the same amount of time, throughput is also improved. A similar case happens with horizontal scaling. By adding more machines and parallelize data computation, the system is immediately able to process more events in the same amount of time, improving throughput, and consequently, reducing the chances of buffering, also potentially improving latency.

3.1.10 Resource Management In stream processing, incoming data are processed on the fly and results are forwarded to other components for further processing. Only a small amount of data, compared to the input rate, would be stored after these analyses. A fundamental problem in such stream processing networks is how to best utilize available resources and coordinate the multiple components so that the overall system performance is optimized.

Nowadays, stream processing systems are loosely coupled with existing resource management systems. In this section, we address the two most common resource management systems used.

Mesos and YARN. Mesos[14] and YARN (Yet Another Resource Negotiator)[15] are currently two Apache open source projects. Mesos was built to be a scalable global resource manager for an entire data center. It was designed at UC Berkeley in 2007 and hardened in production at companies like Twitter and Airbnb. YARN was created out of the necessity to scale Hadoop. Prior to YARN, resource management was embedded in Hadoop MapReduce V1, and it had to be removed in order to help MapReduce scale. The MapReduce V1 JobTracker wouldnt practically scale beyond a couple thousand machines. The creation of YARN was essential to the next iteration of Hadoops lifecycle, primarily around scaling.

Mesos scheduling. Mesos determines which resources are available, and it makes offers back to an application scheduler (the application scheduler and its executor is called a framework). Those offers can be accepted or rejected by the framework. This model is considered a non-monolithic model because it is a two-level scheduler, where scheduling algorithms are pluggable. Mesos allows an infinite number of schedule algorithms to be developed, each with its own strategy for which offers to accept or decline, and can accommodate thousands of these schedulers running multi-tenant on the same cluster.

The two-level scheduling model of Mesos allows each framework to decide which algorithms it wants to use for scheduling the jobs that it needs to run. Mesos plays the arbiter, allocating resources across multiple schedulers, resolving conflicts, and making sure resources are fairly distributed based on business strategy. This model is based on years of operating system and distributed systems research and is very scalable.

YARN scheduling. When a job request comes into the YARN resource manager, it evaluates all the resources available, and it places the job. It is the one making the decision where jobs should go. Thus, it is modeled in a monolithic way. It is important to reiterate that YARN was created as a necessity for the evolutionary step of the MapReduce framework. YARN took the resource-management model out of the MapReduce 1 JobTracker, generalized it, and moved it into its own separate ResourceManager component, largely motivated by the need to scale Hadoop jobs.

YARN is optimized for scheduling Hadoop jobs, which are historically (and still typically) batch jobs with long run times. This means that YARN was not

designed for long-running services, nor for short-lived interactive queries (like small and fast Spark jobs), and while it is possible to have it schedule other kinds of workloads, this is not an ideal model. The resource demands, execution model, and architectural demands of MapReduce are very different from those of long-running services, such as web servers or SOA applications, or real-time workloads like those of Spark or Storm. Also, YARN was designed for stateless batch jobs that can be restarted easily if they fail. It does not handle running stateful services like distributed file systems or databases.

YARN vs. Mesos When comparing YARN and Mesos, it is important to understand the general scaling capabilities and why someone might choose one technology over the other. While some might argue that YARN and Mesos are competing for the same space, they really are not. The people who put these models in place had different intentions from the start, therefore, each approach will yield different long-term results.

When one evaluates how to manage ones data center as a whole, one has Mesos on one side that can manage all the resources in your data center, and on the other, one have YARN, which can safely manage Hadoop jobs, but is not capable of managing an entire data center. Data center operators tend to solve for these two use cases by partitioning their clusters into Hadoop and non-Hadoop worlds. With this in mind, a new project that joins YARN and Mesos came into existence, the Myriad project [16]. In short, Myriad enables Mesos to manage YARN resource requests.

3.2 Approximate Computing

Approximate computing is based on the observation that many data analytics jobs are amenable to an approximate, rather than the exact output. For such an approximate workflow, it is possible to trade accuracy by computing over a partial subset instead of the entire input data to achieve low latency and efficient utilization of resources.

3.2.1 Approximate Computing Techniques Systems that focus in large-scale computing usually rely on Sampling, Histograms, Sketches, Wavelets or in a combination of the above. More specifically, in batch and stream processing solutions, Sampling is the most popular. Since Sampling methods are the most popular, we will now explore some them.

Sampling methods are classified as either probabilistic or non-probabilistic. In probability samples, each member of the population has a known non-zero probability of being selected. Probabilistic methods include random sampling and stratified sampling. In non-probabilistic sampling, members are selected from the population in some nonrandom manner. These include convenience sampling, judgment sampling and quota sampling. The advantage of probability sampling is that sampling error can be calculated. Sampling error is the degree to which a sample might differ from the population. When inferring to the population, results are reported plus or minus the sampling error. In non-probabilistic

sampling, the degree to which the sample differs from the population remains unknown. There are quite a few solutions that use sampling to obtain approximate results, e.g. [17][18][19][20].

Random Sampling. It is the purest form of probability sampling. Each member of the population has an equal and known chance of being selected. When there are very large populations, it is often difficult or impossible to identify every member of the population, so the pool of available subjects becomes biased.

Systematic Sampling This sample often used instead of random sampling. After the required sample size has been calculated, every Nth record is selected from a list of population members. As long as the list does not contain any hidden order, this sampling method is as good as the random sampling method. Its only advantage over the random sampling technique is simplicity. Systematic sampling is frequently used to select a specified number of records from a computer file.

Stratified Sampling. Stratified is superior to random sampling in that it reduces sampling error. A stratum is a subset of the population that share at least one common characteristic. Examples of stratums might be males and females, or managers and non-managers. The researcher first identifies the relevant stratums and their actual representation in the population. Random sampling is then used to select a sufficient number of subjects from each stratum. "Sufficient" refers to a sample size large enough for us to be reasonably confident that the stratum represents the population. Stratified sampling is often used when one or more of the stratums in the population have a low incidence relative to the other stratums.

Convenience Sampling. Convenience sampling is used in exploratory research where the researcher is interested in getting an inexpensive approximation of the truth. As the name implies, the sample is selected because they are convenient. This non-probabilistic method is often used during preliminary research efforts to get a gross estimate of the results, without incurring the cost or time required to select a random sample.

Judgment Sampling. Judgment sampling is a common non-probabilistic method. The researcher selects the sample based on judgment. This is usually an extension of convenience sampling. For example, a researcher may decide to draw the entire sample from one "representative" city, even though the population includes all cities. When using this method, the researcher must be confident that the chosen sample is truly representative of the entire population.

Quota Sampling. This sampling is the non-probabilistic equivalent of stratified sampling. Like stratified sampling, the researcher first identifies the stratums and their proportions as they are represented in the population. Then con-

venience or judgment sampling is used to select the required number of subjects from each stratum. This differs from stratified sampling, where the stratums are filled by random sampling.

SAMPLING ON STREAMS

Data-stream sampling problems require the application of many ideas and techniques from traditional database sampling, but also need new innovations, especially to handle queries over unbounded streams.

Reservoir Sampling. Reservoir sampling is a family of randomized algorithms for randomly choosing a sample of k items from a list or stream S containing n items, where n is either a very large or unknown number. Typically n is large enough that the list does not fit into main memory. The idea is to initialize a reservoir of k elements by inserting elements e1,e2,...,ek until it is full. After the reservoir is full, for i = k + 1, k + 2,...,n, element ei is inserted in the reservoir with a specified probability of k/i. Then, an inserted element overwrites a pre existing element that is chosen randomly and uniformly from the k elements currently in the reservoir. The idea to retain is that it can choose a sample from an unbounded stream of events where the probability of selecting each event is the same. Variations of this algorithm have been experimented where instead of having a static k number as the reservoir size, the sample size is represented as percentage of the data stream seen so far [21] or conveniently varies over time [22].

As the Random Sampling method, this method can be used in conjunction with other sampling method, for instance:

Stratified Reservoir Sampling [23]. As previously mentioned, Stratified Sampling identifies the different stratums. On streams, the goal is usually to achieve an optimal allocation of a fixed-size reservoir to individual sub-streams that may exist. Then, the conventional Reservoir Sampling method is applied to each sub-stream independently. This way, we ensure that all stratums or sub-streams are equally represented in the sample.

Congressional Sampling [24][25]. Congressional sampling is an efficient method of performing sampling when data is partitioned in groups. The algorithm, inspired by the organisation of the United States Congress, where the Congress consists of two differently organised bodies, the House and the Senate, implements a three-stage sampling technique. The first, House, stage allows for item groups to be represented proportionately to their size in the data set (equivalent to Stratified Reservoir Sampling). The second, Senate, stage assigns equal space to each group, while the last stage attempts to even out the sub-group representations in each group. By doing this, the algorithm uses a biased sampling technique at a higher level of the sampling process. On the other hand, each item is sampled with uniform sampling. However, the probability that it will be sampled in the House, Senate or Congress stage is different. Because of this, Congressional sampling is a hybrid of uniform and biased sampling.

3.3 Relevant Related Systems

In this section, we describe some of the most relevant and notorious related systems. We start to describe some stream processing systems such as Storm, Spark and Flink. Next, we describe some solutions based on approximation mechanisms including Blinkdb and Incapprox.

Storm[26][7]. It is a real time analysis engine. Since Storm processes tuple-at-a-time, it can achieve some of the smallest latencies within the stream processing systems. However, it only guarantees that every tuple will be processed at least once. Storm is mostly written in Clojure, and can be used with any programming language. The application is designed as a topology, with the shape of a Directed Acyclic Graph (DAG). Spouts and bolts act as the vertices of the graph.

To accommodate the lack of correctness of Storm, **Trident** was created. Trident is a higher level micro-batching system build atop Storm. It simplifies topology building process and also higher level operations like windowing, aggregations or state management. In addition to Storm's at most once, Trident provides exactly once delivery, on the contrary of Storms at most once guarantee. This comes however, with the cost of added latency.

Spark[27][28]. Apache Spark is a batch processing framework that has the capability of stream processing, as well, making it a hybrid framework. Spark offers a declarative API and is most notably easy to use, and it is easy to write applications in Java, Scala and Python. This open-source cluster-computing framework is ideal for machine-learning, but does require a cluster manager and a distributed storage system. Furthermore, it offers exactly once guarantees.

Spark relies on a data structure known as the Resilient Distributed Dataset (RDD). This is a read-only multiset of data items that is distributed over the entire cluster of machines. RDDs operate as the working set for distributed programs, offering a restricted form of distributed shared memory.

Spark is one of the most popular engine used for streaming and one of the most complete, if not for the fact that it processes in mini-batches, which limits its ability to provide low-latency results.

Flink[29][30]. Flink is a newer framework that, just like Spark, offers both stream and batch capabilities thus, making it a hybrid framework. However, while Spark was initially designed as for batch processing, Flink was designed for stream processing from scratch. In Flink, batch processing is seen as a subset of stream processing (linked streams). This allows for high throughput, low latency computations that are written in Java, Scala, Python, and SQL, on a declarative API.

Flinks applications are all fault-tolerant and can support exactly once semantics. It saves its sate through lightweight snapshots[31][32], and is one of the most complete systems around.

BlinkDB[18]. BlinkDB is a massively parallel, approximate query engine for running interactive SQL queries on large volumes of data. BlinkDB allows users to trade-off query accuracy for response time, enabling interactive queries over massive data by running queries on data samples and presenting results annotated with meaningful error bars. To achieve this, BlinkDB uses two key ideas: 1) an adaptive optimization framework that builds and maintains a set of multi-dimensional stratified samples from original data over time, and 2) a dynamic sample selection strategy that selects an appropriately sized sample based on a querys accuracy or response time requirements.

IncApprox[17]. IncApprox is a data analytics system based on Spark Streaming. Its core logic is based on the marriage of approximate computing with incremental computing. Their main idea is to create a sampling algorithm that biases the sample selection to the memoized data items from previous runs. They rely on memoization of intermediate results of sub-computations, and try to reuse these memoized results across jobs.

Analysis and discussion

After the description of the related work throughout this section, we now present a table (Table 1) that shows a comparative analysis of the systems we have studied.

Performance was based on architectural choices such as the stream model and state storing, e.g., Flink allows for low latency with its true stream model, while its lightweight asynchronous snapshots mechanism allows for the system to scale without affecting its throughput too much. The other features and criteria on the table are those studied in this section.

4 Proposed solution

Taking into consideration the analysis of the various systems presented in Table 1, we chose Flink as the open source system that well will use to implement our solution. Even though there are more mature systems such as Storm and Spark, Flink is gaining popularity very fast. Furthermore, it combines low latency, which Storm provides, with high throughput, which Spark provides[37][38].

Our proposed solution consists of an extension to Flink that will allow users to trade-off accuracy of results for response time. To achieve this, we will use sampling algorithms and also combinations of sampling algorithms so we can run the program on a subset of data enough to guarantee the user specified requirements.

In the following sections, we will explain in more detail how we will accomplish this. In Section 4.1 we detail our architecture and in Section 4.2 we describe our main sampling algorithms.

4.1 Architecture Overview

Now we are going to describe in more detail, the new added components inside Flink, and how they interact with each other. The architecture is depicted on Figure 8.

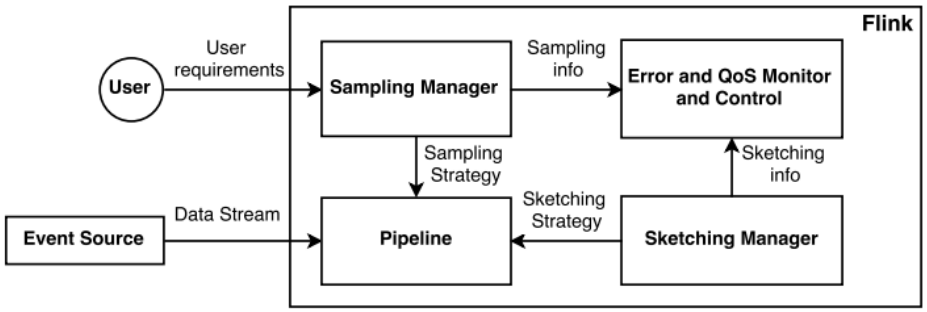

Fig. 8. Solution Architecture Overview

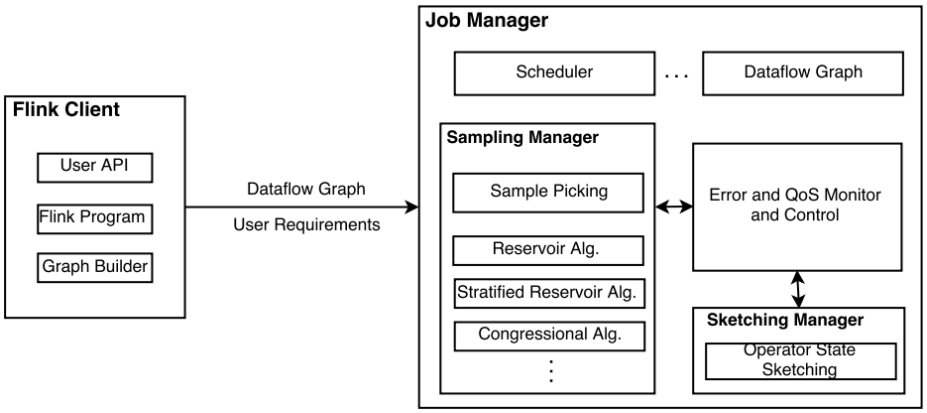

Fig. 9. Solution Software Architecture

In our solution, as in Figure 8, the user specifies his query requirements in terms of latency or accuracy, which will be received by the Sampling Manager component. This is the main component of our solution which is currently not present in base Flink. This component will employ the user requirements to determine the sampling size to meet the expected requirements, as an adaptive process. It will also analyze the pipeline and identify what best sampling algorithm is fit for the operations at hand e.g. if the prevalent function is to

calculate an average, use a simple and fast uniform sampling algorithm since the data distribution is not very important. After the component calculates the sample size and figures out what the sampling plan should be used, it sends the sampling strategy to the operations at hand. It also sends the necessary sampling information to the Error and QoS Monitor.

We also propose a Sketching Manager whose goal is to save a sketch of each operator's state, as in Figure 9, so we can later on perform operations on them.

The Error and QoS Monitor component will monitor the pipeline performance and its results and will use the information provided by the previous component to calculate the errors associated with the sampling strategy used. (Error component depois reporta o erro e regula o sampling consoante o erro e a velocidade? e depois o sample managing vai corrigindo o sampling?)

4.2 Main Sampling Algorithms

Here we present some of the main sampling algorithms that will be used in our solution.

Algorithm 1 Conventional Reservoir Sampling (CRS)

Require $|r|$ *// the size of the reservoir r*
1: $r = rSize(userReq)$; *// estimate the reservoir size to meet user requirements*
2: $k = 0$;
3: **for** each tuple arriving from the input stream **do**
4: $k = k + 1$;
5: **if** $k \leq |r|$ **then**
6: add the tuple to the reservoir
7: **else**
8: decide with the probability $\frac{|r|}{k}$ whether to accept the tuple
9: **if** the tuples are accepted **then**
10: replace a randomly selected tuple in the reservoir with the accepted tuple
11: **end if**
12: **end if**
13: **end for**

5 Evaluation Methodology

In this section we present the metrics and some benchmark applications that can be used to evaluate our work.

Key Metrics:

- Speed-up in processing time, which can be partitioned in:

System	Streaming Model	Programing Model	Window Management	Guarantees	State Management	Performance	API Language
Storm[26]	Stream	Compositional	Time-based Count-based	Exactly-once	Record ACKs[2]	Latency oriented	Java/Scala/ Python
Storm Trident [26]	Micro-batch	Compositional	Time-based Count-based	At least once	Record ACKs	Trades latency for correctness	Java/Scala/ Python/SQL
Spark Streaming[27]	Micro-batch	Declarative	Time-based Count-based	Exactly once	Checkpoint with RDDs	Throughput oriented	Java/Scala/ Python/SQL
Flink [29]	Stream	Declarative	Time-based Count-based	Exactly once	Distributed snapshots	Latency and throughput oriented	Java/Scala/ Python/SQL
Samza [33][34]	Stream	Compositional	Time-based	At least once	Local and distributed snapshots	Latency and throughput oriented	Java
Heron [35]	Stream	Compositional	Time-based Count-based	At least once	Record ACKs	Latency oriented	Java/Scala/ Python
Google cloud dataflow [36]	Stream	Declarative	Time-based Count-based	Exactly once	Checkpoint all records + ACKs	Latency and throughput oriented	Java/Python

Table 1. Systems Comparison

Algorithm 2 Congressional Sampling

```
initialize(sample, group);
sampleCount ← 0;
houseSample ← 0;
senateSample ← 0;
groupingSample ← 0;
for all item ∈ dataStream do
    doHouseSample(item)
    doSenateSample(item)
    for attribute ∈ group do
        doGroupingSample(item)
    end for
end for
getFinalCongressionalGroups(groupingSample)
calculateSlots(houseSample, senateSample, groupingSample)
scaleDownSample()
```

  - Input Rate - Input rate at which tuples are injected to the input streams
  - Throughput - Observe how applications keep up with the input rate
  - Latency - Time for tuples to be processed

- Resource usage, which can be partitioned in:
  - Used CPU - Percentage of CPU being used by the nodes/tasks
  - Used memory - Percentage of memory being used by the nodes/tasks
  - Variation in memory consumption.

- Accuracy of results, which can be partitioned in:
  - Error metric, estimating the relative error in the generated sample of the benchmark applications.
  - Accuracy per Query - Observe how the accuracy of the queries varies over time

Bechmark Applications. Besides micro-benchmarks to exercise various statistic distributions of tuples to validate and assess the sampling and skecthing mechanisms.

- NASDAQ Tweets - Benchmark application based on example provided from UC Berkeley AMPLab website. It performs an analysis over a data set of Apple NASDAQ tweets. Real-time Processing with Spark Streaming[39]; FollowTheHashTag, One hundred NASDAQ 100 Companies Free Twitter Datasets[40]
- US Technology Companies Stock, based on an example of a stock analysis application. It performs data analysis over a data set of the US stock market.
- James Phillpotts, Real-time Data Analysis Using Spark[41]
- Quandl, Wiki EOD Stock Prices[42]

6 Conclusion

In this work we have presented the state of the art of stream processing. We started by presenting the motivation that has led us to make this work, also defining its objectives. We then introduced the key concepts of stream processing along with the main systems used for this purpose. After discussing the current situation, we described our solution. Finally we presented the metrics used to evaluate our solution and assess the quality of the work developed.

7 Work scheduling

References

1. S. Chang, Y. Zhang, J. Tang, D. Yin, Y. Chang, M. A. Hasegawa-Johnson, and T. S. Huang, "Streaming recommender systems," in *Proceedings of*

Tasks	Details	Duration
Preparation	- Further study Flink - Setup development environment - Find/make good use case examples - Integration with external systems	January (1 week) February (2 weeks)
Implementation	- Setup and implement components of the architecture - Data structures implementation - Implement algorithms	February (2 week) March (4 weeks) April (4 weeks)
Last touch-ups	- Conclude any unimplemented functionality - Bugs detection and fix	May (4 weeks)
Metrics measurements	- Setting up test environment - Collect/make more use cases to test and benchmark the solution - Evaluate implemented solution	June (4 weeks) July (2 weeks)
Thesis final report writting	- Write final thesis report	July (2 weeks) August (4 weeks)
Review and submission	- Report review and submission	September (2 weeks)
Documentation	- Document design choices, code and tests	January - September
Bi-weekly meetings	- Analyze work progression	January - September

Table 2. Work scheduling

the 26th International Conference on World Wide Web, ser. WWW '17. Republic and Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee, 2017, pp. 381–389. [Online]. Available: https://doi.org/10.1145/3038912.3052627

2. "CEP Patterns for Stream Analytics," https://dzone.com/articles/complex-event-processing-for-stream-analytics, accessed: 2018-01-02.
3. S. Guha, N. Mishra, G. Roy, and O. Schrijvers, "Robust random cut forest based anomaly detection on streams," in *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ser. ICML'16. JMLR.org, 2016, pp. 2712–2721. [Online]. Available: http://dl.acm.org/citation.cfm?id=3045390.3045676
4. "The Real-Time Future of ETL," http://sqlstream.com/2017/05/real-time-future-etl/, accessed: 2018-01-02.
5. "Real-time Twitter sentiment analysis in Azure Stream Analytics," https://docs.microsoft.com/en-us/azure/stream-analytics/stream-analytics-twitter-sentiment-analysis-trends, accessed: 2018-01-02.
6. "Questioning the Lambda Architecture," https://www.oreilly.com/ideas/questioning-the-lambda-architecture, accessed: 2018-01-02.
7. "Apache Storm," http://storm.apache.org/, accessed: 2018-01-02.
8. "Apache Storm," http://hadoop.apache.org/, accessed: 2018-01-02.
9. J. Kreps, N. Narkhede, and J. Rao, "Kafka: a distributed messaging system for log processing," 2011.
10. T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle, "The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," *Proc.*

VLDB Endow., vol. 8, no. 12, pp. 1792–1803, Aug. 2015. [Online]. Available: http://dx.doi.org/10.14778/2824032.2824076
11. Q. To, J. Soto, and V. Markl, "A survey of state management in big data processing systems," *CoRR*, vol. abs/1702.01596, 2017. [Online]. Available: http://arxiv.org/abs/1702.01596
12. "Fault tolerance for stream processing engines," *CoRR*, vol. abs/1605.00928, 2016, withdrawn. [Online]. Available: http://arxiv.org/abs/1605.00928
13. X. Chen, Y. Vigfusson, D. M. Blough, F. Zheng, K.-L. Wu, and L. Hu, "Governor: Smoother stream processing through smarter backpressure," *14th IEEE International Conference on Autonomous Computing*, Jul 2017. [Online]. Available: http://par.nsf.gov/biblio/10048564
14. B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 295–308. [Online]. Available: http://dl.acm.org/citation.cfm?id=1972457.1972488
15. V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13. New York, NY, USA: ACM, 2013, pp. 5:1–5:16. [Online]. Available: http://doi.acm.org/10.1145/2523616.2523633
16. "Apache Myriad," http://myriad.apache.org/, accessed: 2018-01-02.
17. D. R. Krishnan, D. L. Quoc, P. Bhatotia, C. Fetzer, and R. Rodrigues, "Incapprox: A data analytics system for incremental approximate computing," in *Proceedings of the 25th International Conference on World Wide Web*, ser. WWW '16, 2016, pp. 1133–1144. [Online]. Available: https://doi.org/10.1145/2872427.2883026
18. S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica, "Blinkdb: Queries with bounded errors and bounded response times on very large data," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13. New York, NY, USA: ACM, 2013, pp. 29–42. [Online]. Available: http://doi.acm.org/10.1145/2465351.2465355
19. I. Goiri, R. Bianchini, S. Nagarakatte, and T. D. Nguyen, "Approxhadoop: Bringing approximations to mapreduce frameworks," *SIGARCH Comput. Archit. News*, vol. 43, no. 1, pp. 383–397, Mar. 2015. [Online]. Available: http://doi.acm.org/10.1145/2786763.2694351
20. S. Kandula, A. Shanbhag, A. Vitorovic, M. Olma, R. Grandl, S. Chaudhuri, and B. Ding, "Quickr: Lazily approximating complex adhoc queries in bigdata clusters," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16. New York, NY, USA: ACM, 2016, pp. 631–646. [Online]. Available: http://doi.acm.org/10.1145/2882903.2882940
21. T. R. Kepe, E. C. de Almeida, and T. Cerqueus, "Ksample - dynamic sampling over unbounded data streams," 2015.
22. M. Al-Kateb, B. S. Lee, and X. S. Wang, "Adaptive-size reservoir sampling over data streams," in *SSDBM*, 2007.
23. M. Al-Kateb and B. S. Lee, "Stratified reservoir sampling over heterogeneous data streams," in *Proceedings of the 22Nd International Conference on Scientific and Statistical Database Management*, ser. SSDBM'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 621–639. [Online]. Available: http://dl.acm.org/citation.cfm?id=1876037.1876087

24. N. Koevski, "Advanced sampling in stream processing systems," Portugal, Lisbon, 2016. [Online]. Available: https://fenix.tecnico.ulisboa.pt/downloadFile/1689244997256259/Extended-Abstract-Nikola-Koevski.pdf
25. S. Acharya, P. B. Gibbons, and V. Poosala, "Congressional samples for approximate answering of group-by queries," in *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '00. New York, NY, USA: ACM, 2000, pp. 487–498. [Online]. Available: http://doi.acm.org/10.1145/342009.335450
26. A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy, "Storm@twitter," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14. New York, NY, USA: ACM, 2014, pp. 147–156. [Online]. Available: http://doi.acm.org/10.1145/2588555.2595641
27. M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: ACM, 2013, pp. 423–438. [Online]. Available: http://doi.acm.org/10.1145/2517349.2522737
28. "Spark," https://spark.apache.org/, accessed: 2018-01-02.
29. P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *IEEE Data Eng. Bull.*, vol. 38, pp. 28–38, 2015.
30. "Flink," https://flink.apache.org/, accessed: 2018-01-02.
31. P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas, "State management in apache flink®: Consistent stateful distributed stream processing," *Proc. VLDB Endow.*, vol. 10, no. 12, pp. 1718–1729, Aug. 2017. [Online]. Available: https://doi.org/10.14778/3137765.3137777
32. P. Carbone, G. Fóra, S. Ewen, S. Haridi, and K. Tzoumas, "Lightweight asynchronous snapshots for distributed dataflows," *CoRR*, vol. abs/1506.08603, 2015. [Online]. Available: http://arxiv.org/abs/1506.08603
33. S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell, "Samza: Stateful scalable stream processing at linkedin," *Proc. VLDB Endow.*, vol. 10, no. 12, pp. 1634–1645, Aug. 2017. [Online]. Available: https://doi.org/10.14778/3137765.3137770
34. "Samza," http://samza.apache.org/, accessed: 2018-01-02.
35. S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter heron: Stream processing at scale," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: ACM, 2015, pp. 239–250. [Online]. Available: http://doi.acm.org/10.1145/2723372.2742788
36. T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle, "The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1792–1803, Aug. 2015. [Online]. Available: http://dx.doi.org/10.14778/2824032.2824076
37. M. A. Lopez, A. G. P. Lobato, and O. C. M. B. Duarte, "A performance comparison of open-source stream processing platforms," in *GLOBECOM*, 2016.

38. S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, and P. Poulosky, "Benchmarking streaming computation engines: Storm, flink and spark streaming," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2016, pp. 1789–1792.
39. "Real-time Processing with Spark Streaming," http://ampcamp.berkeley.edu/3/exercises/realtime-processing-with-spark-streaming.html, accessed: 2018-01-02.
40. "One hundred NASDAQ 100 Companies Free Twitter Datasets," http://followthehashtag.com/datasets/nasdaq-100-companies-free-twitter-dataset/, accessed: 2018-01-02.
41. "Real-time Data Analysis Using Spark," http://blog.scottlogic.com/2013/07/29/spark-stream-analysis.html, accessed: 2018-01-02.
42. "Wiki EOD Stock Prices," https://www.quandl.com/data/WIKI/documentation/bulk-download, accessed: 2018-01-02.