STREAMLINE. FAST REACTIVE ANALYTICS

D1.1 - Combined Data at Rest and Data in Motion Analysis Platform (Revised)

Project Number	688191
Project Acronym	STREAMLINE
Nature	R: Report
Dissemination Level	Public
Work Package	WP1
Due Delivery Date	November 2016
Actual Delivery Date	November 2016
Revision Delivery Date	November 2017
Lead Beneficiary	DFKI
Authors	Quoc-Cuong To Behrouz Derakhshan Tilmann Rabl

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 688191.

Executive Summary

This deliverable describes a unified processing system based on the requirements described in the STREAMLINE project, which enables the user to combine and process data in motion (aka data streams) and data at rest (aka historical data). Since the requirements of the system are based on the needs of the use cases of the STREAMLINE industrial partners, the resulting processing engine will enable the STREAMLINE partners to overcome the shortcomings of their current processing system. The requirements of the unified batch and stream platform of the STREAMLINE project are as follows:

- Efficiency: the proposed unified computing model must be as efficient as the model for separate stream or batch.
- Maintainability: the proposed unified model must be easy to maintain.
- Optimization: a new optimization technique to incorporate both kinds of data is required
- Fault Tolerance: the unified stream and batch processing must be designed in a fault-tolerant model.
- Incremental computation: incoming data needs to be processed incrementally to speed up the computation by evolving the result over incoming data changes in a stream.

We start this deliverable by describing the architecture of Apache Flink, a big data platform with separate APIs for batch and stream data. We build the unified platform of STREAMLINE on Apache Flink.

We start Section 3 by analyzing the shortcomings of existing computing model of the STREAMLINE project partners. Then, we analyze the differences between DataSet and DataStream APIs in Apache Flink. Based on the shortcoming of the current systems and what Flink APIs offer, we point out the requirements that the unified system supporting data at rest and data in motion must satisfy.

In Section 4, we list the existing optimizations in literature and show how we can extend these optimizations to apply to dataflow graph containing operators both on data at rest and data in motion. We present a new load shedding optimization technique for parallel data flows, including batch and streaming data sources, based on M4 aggregation technique which performs early data reduction and can be used for interactive data processing and visualization.

In Section 5, we present the requirements for state management and fault tolerance. Fault tolerance requires taking consistent snapshot of the state for the checkpoint and recovery.

Finally, we define the concept of User Defined Windows (UDFs) and incremental aggregations in Section 6. We introduce the Cutty framework that supports a strategy to incrementally pre-compute higher-level aggregates and share them between overlapping windows.

Summary of Revision

The following table shows the revision comments that we have received and how we fix them, e.g, by providing clarification, adding new paragraphs or sections in the specified places of the deliverable.

	Revision Comment	**Actions**
1	The first version of the platform is not yet available	The link to the main Streamline platform was added to the deliverable in ***Section 3.5***.
2	Comment on the delay in PR	Initially, we planned to merge Streamline improvements into Apache Flink using Pull Requests (PRs). However, as it is explained in Flink guideline for Pull Request Management the process of approving new complex feature such Side Input can be quite lengthy, we have decided to push Streamline improvements to a fork of Flink. We explained this in details in ***Section 3.5***.
3	Provide information on the optimization technique chosen	We added information about the M4 optimization that was implemented in Streamline in ***Section 4.3.***
4	Provide all the relevant publications (+ text accessibility)	All relevant publications have been added to the text as references and are available from the Streamline website: http://h2020-streamline-project.eu/publications/
5	Provide and make sure that all links to repos are working	We added links to all code in the Streamline repository. All sub-repositories can be found here: https://github.com/streamline-eu/
6	If there is a change in the approach, and the platform will be delivered as an addition to the Flink, then this should be described in the Deliverable	As we explained in the Actions section of the **comment 2** and later in ***Section 3.5***, we have decided to push all Streamline improvements to Flink to a fork of Flink on Streamline platform. The reason is that pull request management of Flink for new complex features can be a quite lengthy process and the decision is not in our hands.

Table of Contents

List of Figures

List of Tables

List of Abbreviations and Acronyms

ICT	Information and Communication Technologies

1 Introduction

In big data setups, the most effective way to process very large amounts of data collected over a longer period of time is using batch data processing. Data is collected and processed in batch (e.g., by using Apache Hadoop [1,2] or Apache Spark [15] as the most popular open source system for batch processing) and then the final result is produced at once. The solution to process newly arriving data is to run periodic jobs on these data. Frameworks for batch processing try to improve data throughput as their first priority (i.e., process as much data as they can in a unit of time) and often ignore other metrics, like latency. Therefore, they cannot be used in real-time applications in which very low latency is required.

Indeed, real time data processing is an emerging area and can be seen in every aspect of life (e.g., from stock market data, to news on televisions). In these real time applications, the input data arrives at the processing system at very high rate and, therefore, a mechanism to process these fast data efficiently and with low latency is required. Simply storing these fast data on disk and then processing it in a way similar to batch processing does not work because data must be processed in a small period of time (or near real time) to meet the latency requirement. Failing to meet this latency requirement prevents users to take immediate action to adapt to the fast changing events embedded in these data. Stream processing frameworks (e.g., Storm, Flink) were born to meet this requirement by providing fast and reliable processing capability.

Although these frameworks can provide batch processing with high throughput and stream processing with low latency, it still remains an open problem how to combine batch and stream processing into a single system in a seamless fashion to ensure simplicity, maintainability, efficiency, and fault tolerance. This is one of the objectives of the STREAMLINE project and this document points out the problems and define requirements to reach this objective.

The need to combine both batch and stream data due to its benefit is emerging in many applications and unified (i.e., streaming and batch) data processing is gaining importance. While batch data is valuable in providing insight knowledge of historical events, it cannot reflect the fresh and up-to-date knowledge of data streams. Therefore, combining these two kinds of data to enjoy the benefits of both worlds is necessary. Indeed, there are many applications of such unified system from media and game content business (e.g., NMusic, Rovio) to web scale data extraction (e.g., IMR). To implement such a system, there are requirements to be considered. Table 1 lists the important requirements for the unified system. Each requirement will be discussed in more detail in the following sections.

Efficiency	We process both stream and batch data in real-time applications. Meeting the low latency requirement of stream processing to ensure the efficiency of the system is essential. Furthermore, to handle massive amounts of data, the processing time should also be low (with the proportional increase of computing resource) to reflect the overall efficiency of the unified system.
Maintainability and ease of programming	The second requirement for the unified system is that it must be easy to program the combined batch and stream data analytics, avoiding the complexity of programming and maintaining the codes for two separate systems. This requirement can be achieved by using high level unified APIs. This requires that when there is a change in the program logic in the future, the amount of code changes is minimal.
Optimization	Currently, optimization techniques can only be applied to either batch or stream data processing systems. One example is that fast and big data systems cannot optimize joint operations on data at rest and data in motion. So we have to find out how we can apply various optimization opportunities on big and fast data by leveraging the properties of both kinds of data.
Fault Tolerance	Due to the massive amount of batch and stream data in this unified system, deploying the processing system to multiple distributed nodes to ensure the scalability is necessary. This can lead to the higher failure probability in the computing nodes due to the increasing number of nodes. Therefore, availability of such system is essential. Satisfying this requirement ensures the seamless computation of the unified system in case of failure.
Incremental computation	Currently, Flink supports only delta iterations [5] that can update a mutable state carried forward to the next iteration. We extend this concept to enable incremental computation and sharing of partial results which evolve with arrival of new elements of streams.

Table 1. General requirements for hybrid processing system.

1.1 Document objectives

To build the unified data processing system, there are some requirements and challenges to be dealt with. The requirements and challenges defined for the unified architecture are:

- Simple, efficient, and maintainable computing system,

- High level API for unified processing with common operators for both batch and stream data,
- Optimization,
- Fault tolerance,
- Incremental computation.

1.2 Document structure

The rest of the document can be summarized as follows. We first present Apache Flink [4] in Section 2, because it is the basis of our implementation of the hybrid computational model. In Section 3, we differentiate the DataSet and DataStream APIs of Flink to clarify requirements for our high level API and computing platform. Furthermore, we analyze their shortcomings for the business use cases of the STREAMLINE project partners. Based on these analyses, we point out the requirements that our proposed unified system for combining data at rest and data in motion must satisfy. Next, we present the query optimization engine in Section 4. After that, we discuss the fault tolerance in Section 5. Then, the incremental computation will be presented in section 6. Finally, we conclude this document by giving the main points of requirements of proposed system.

2 Apache Flink Overview

This section overviews the Flink architecture to see the differences between Stream and Batch processing in this framework. Then, we provide an example on how Flink currently processes batch and stream data.

2.1 Architecture

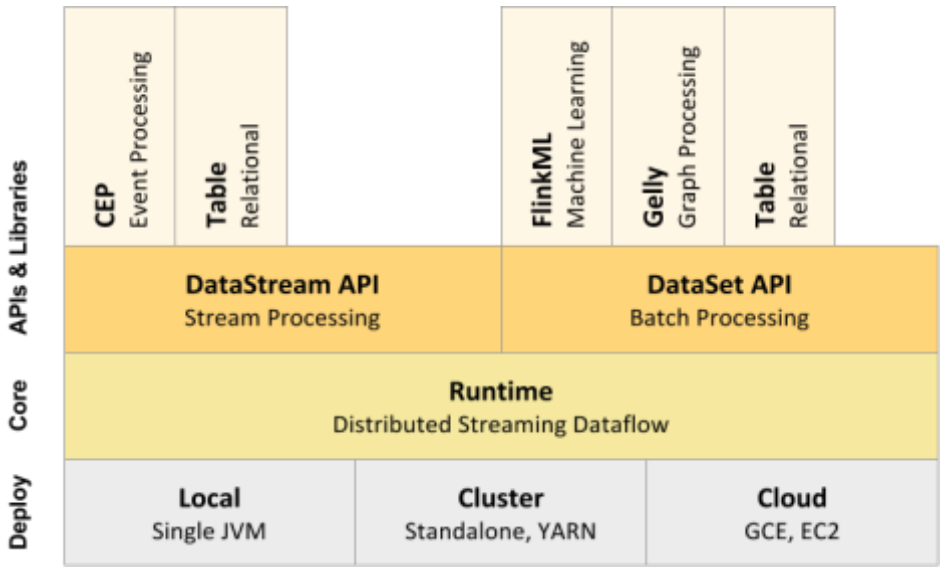


Figure 1. Architecture of Apache Flink.

The core of Apache Flink is a streaming dataflow engine that provides data distribution, communication, and fault tolerance for distributed computations over data streams.

Flink includes several APIs for creating applications that use the Flink engine:

- DataStream API for unbounded streams embedded in Java and Scala, and
- DataSet API for static data embedded in Java, Scala, and Python,
- Table API with a SQL-like expression language embedded in Java and Scala.

Flink also bundles libraries for domain-specific use cases:

- CEP, a complex event processing library,
- Machine Learning library, and
- Gelly, a graph processing API and library.

Figure 1 sketches the architecture of Apache Flink [3,4]. It can be easily seen that Flink provides separate APIs for batch and stream processing. Although Flink has only one runtime engine in which all programs are executed, users have to write programs to process batch or stream data separately using the corresponding APIs. Depending on the type of data sources (i.e., batch or stream data), the user need to write either a batch program or a streaming program using the DataSet API for the former and the DataStream for the latter. Flink has the special classes for DataSet and DataStream to represent data in a program. One can think of the APIs as immutable

collections of data that can contain duplicates. However, the difference between the two APIs is that the data in the case of DataSet is finite while for a DataStream it can be unbounded.

DataSet programs in Flink are regular programs that implement transformations on data sets (e.g., filtering, mapping, joining, and grouping). The data sets are initially created from certain sources (e.g., by reading files, or from local collections). On the contrary, DataStream programs in Flink implement transformations on data streams (e.g., filtering, updating state, defining windows, and aggregating). The data streams are generated by various sources (e.g., message queues, socket streams, files). Results are returned via sinks, which may for example write the data to (distributed) files, or to standard output (e.g., command line terminal).

Flink is compatible with various cluster management and storage solutions, such as Apache Tez, Apache Kafka [10], Apache HDFS [11], and Apache Hadoop YARN [12].

2.2 Libraries and Interfaces

Apache Flink users can use various programming languages (e.g., Java and Scala) to write their programs. In each language, the corresponding APIs are provided to process either stream or batch respectively. All APIs provide the user with operators such as Map, Reduce, Join, Cross, and Filter. Beside that, users can specify arbitrary user-defined functions.
Listing 1 shows a complete, working example of WordCount written in Java DataSet API. Analogous to this example is the Scala version with shorter lines of code in Listing 2.

```
public class WordCountExample {
    public static void main(String[] args) throws Exception {
        final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

        DataSet<String> text = env.fromElements(
            "Who's there?",
            "I think I hear them. Stand, ho! Who's there?");

        DataSet<Tuple2<String, Integer>> wordCounts = text
            .flatMap(new LineSplitter())
            .groupBy(0)
            .sum(1);

        wordCounts.print();
    }

public static class LineSplitter implements FlatMapFunction<String, Tuple2<String,
Integer>> {
        @Override
        public void flatMap(String line, Collector<Tuple2<String, Integer>> out) {
            for (String word : line.split(" ")) {
                out.collect(new Tuple2<String, Integer>(word, 1));
            }
        }
    }
}
```

Listing 1. Word count example in Flink's DataSet API using Java.

```scala
object WordCount {
  def main(args: Array[String]) {

    val env = ExecutionEnvironment.getExecutionEnvironment
    val text = env.fromElements(
      "Who's there?",
      "I think I hear them. Stand, ho! Who's there?")

    val counts = text.flatMap { _.toLowerCase.split("\\W+") filter { _.nonEmpty } }
      .map { (_, 1) }
      .groupBy(0)
      .sum(1)

    counts.print()
  }
}
```

Listing 2. Word count example in Flink's DataSet API using Scala.

Listing 3 is an example of streaming window word count application written in Scala, that counts the words coming from a web socket in 5 second windows.

```scala
object WindowWordCount {
  def main(args: Array[String]) {

    val env = StreamExecutionEnvironment.getExecutionEnvironment
    val text = env.socketTextStream("localhost", 9999)

    val counts = text.flatMap { _.toLowerCase.split("\\W+") filter { _.nonEmpty } }
      .map { (_, 1) }
      .keyBy(0)
      .timeWindow(Time.seconds(5))
      .sum(1)

    counts.print

    env.execute("Window Stream WordCount")
  }
}
```

Listing 3. Word count example in Flink's DataStream API using Scala.

It is easy to see that the current version of Flink does not allow the interaction between DataSet and DataStream. Section 3 shows the differences between DataSet and DataStream in more detail.

3 Hybrid Data Analysis over Historical and Streaming Data

In this section, we present our findings for Task 1.3 (i.e., Unified Batch-Stream-Processing System) of Work Package 1.
First, we present the requirements for hybrid computing model for high volume and rate data. Then, we introduce Side Input, an abstraction over the data streaming API of Apache Flink that enables combined batch and stream data processing.
To define the requirements for the hybrid system, we proceed as follows:

- We first list the project partner use cases, the problem with their computing platforms and then summarize the requirements in the use cases.
- We review the current state of Flink to see which current features can be extended to the hybrid system to satisfy the requirement sketched in the previous step.

3.1 Project Partner Use Case Requirements

We review the four use cases from the project partners (one use case per partner) and identify the current problems they are facing in their business models, the shortcomings of their current technologies, and then list the requirements for the hybrid system which can solve these problems and challenges.

Quadruple-play business case. Portugal Telecom (ALB) plays the role of national-level leading provider of quadruple play service (i.e., landline, mobile phone, internet, IPTV). Currently, ALB developed in-house tools for analyzing and visualizing the high-throughput stream, with infrastructure consisting of a hodgepodge of technologies, including: (i) batch processing of aggregation and behavior analytics using Hadoop, (ii) Esper as open source Complex Event Processing (CEP) engine, (iii) Stream processing with Storm, (iv) MoneDB/PostgreSQL for ad-hoc analysis, and (v) custom code to glue all of the previous technologies together. With this infrastructure, ALB is not able to efficiently give real-time responses to high throughput user activity streams. Also, in using various platforms linked using the glue code, ALB incurs high code maintenance overhead which is due to the complexity of combining these platforms.
These difficulty in ALB has led to the requirement that the overall system must have a common platform for cross-company data integration. This system must ensure high throughput and low latency in processing both batch and stream data. Also, the other requirement is that the code maintenance must be minimum. That is one source code can be used for either batch or stream or both of them. This leads to the maintainability requirement for a unified system.

Media content business case. NMusic provides streaming platforms for both music and video content. Its computing platform is a lightweight set of components written in C/C++ languages and Ruby On Rails. Currently, the following technologies are used in NMusic: PostgreSQL and MongoDB to hold content catalogs and system activity logs, REDIS/Memcache for frequently accessed static data, Lucene/SOLR search engine, and NGINX web servers to deliver media content via progressive HTTP download.
With the current technologies deployed in their platform, all content and user-related events are processed in batch mode, causing the inability to deliver real-time context-aware content

recommendations to create a personalized service experience. Furthermore, NMusic cannot cold-start their recommendations based on context information (e.g., geo-location, social media).
With these difficulties, NMusic requires a hybrid system to incorporate the processing of stream data into the traditional batch processing so that the up-to-date information on the streaming data can improve the personalized service to attract more customers to their business.

Game and Entertainment business case. Rovio is a famous global brand in gaming, processing over three billion events per day from 30 games. Rovio games have a native SDK used to access Hatch cloud services and to send analytics events. Raw dataset is aggregated to daily level by Hive and scalding jobs and stored into Amazon Redshift database. Scheduled Redshift queries are used to analyze the data. Insight is stored into an Amazon RDS database for data visualization tools and an internal dashboard built on Highcharts library.
Currently, all analytics events are processed in batch only. If a service requires real-time processing, it is implemented into service itself with custom counters and libraries. As a result, the analytics insight is only received the next business day and there is no standard way to implement real-time analysis. Rovio requires a single unified computing platform that can provide both batch and real-time processing. The low latency requirement is essential because currently the user profile is updated on a daily basis with batch process and they want to achieve a profile that has more up-to-date information.

Web scale data extraction. IMR has developed a huge crawling infrastructure with over one million sources updated and a global map of the web created every two months. Currently, IMR uses its in-house developed crawler (MemoryBot written in Erlang), which scales linearly to tens of billions of pages. HBase is used for time-based representation for large datasets. Elastic Search is used for shared index. The coordination of jobs for processing unstructured content is done on Mignify, built on top of Hadoop and Flink.

IMR needs an easy-to-use distributed library for complex tasks such as multilingual event detection, crawl prioritization, and user session analysis. In other words, it requires a simple API from the hybrid system for both big and fast data. Furthermore, it requires an efficient data streaming libraries that can be integrated with MemoryBot for very fast reaction crawl time analytics. Again, the requirement of low latency is important in this case.
Based on these observations of the pain points of partner business case, we review the differences between DataSet and DataStream in Flink in the next section to see how they can be combined into a single system.

3.2 Current state of Flink

To define the requirements for the hybrid system, these differences are important to decide which main functionalities from DataStream and DataSet will be supported in the hybrid system. So before sketching out the requirement for the hybrid system, we analyze these differences in detail.
DataStream programs in Flink are regular programs that implement transformations on data streams (e.g., filtering, updating state, defining windows, aggregating). The data streams are initially created from various sources (e.g., message queues, socket streams, files). On the

contrary, DataSet programs in Flink implement transformations on data sets (e.g., filtering, mapping, joining, grouping). The data sets are initially created from certain sources (e.g., by reading files, or from local collections). In the context of STREAMLINE, the data streams and data sets comes from project partners (i.e., IMR, Rovio, NMusic, ALB).

Currently, data transformations transform one or more DataStreams into a new DataStream, or one or more DataSets into a new DataSet. Although programs can combine multiple transformations into sophisticated assemblies, it is impossible to combine both batch and stream data into a single program by using a mixture of DataSet and DataStream transformations simultaneously because of the differences between these two kinds of data.

There are many differences between DataSet and DataStream, ranging from high level differences to the differences at operator level, optimization techniques, state management, scheduling and fault tolerance.

On a high level, DataSet works with static files and finite inputs, it provides high throughput but also high latency as compared to DataStream's low latency and infinite input data types (Event Streams). DataSet supports both blocking and pipelined data transfer and caches intermediate results, whereas DataStream only supports pipelined data transfer and does not store any intermediate results, but it does however, allow operator states to be persisted throughout the processing. DataSet primarily works with blocking operators with the exception of projection, map, and flatmap (and certain types of aggregation). This is in contrast to DataStream, where most of the operators are non-blocking except for physical partitioning, window joins, and reduce/fold on WindowedStream. This property of the DataStream enables it to process data in real-time.

DataSet and DataStream also have differences in the set of graph translation and optimizations that are performed prior to execution of a batch or streaming job. Join strategy selection, data locality and cost-based optimization are only implemented for DataSet and window pre-aggregation is specific to DataStream. Operator chaining/fusion and iteration head+tail collocation are two of the optimizations that they both share.

Run time optimization and task level semantics are also different for DataSet and DataStream. Features specific to DataStream are back-pressure, barriers, and blocking channels (aligning). DataSet offers eager memory allocation and intermediate result storage, two features that are currently missing from DataStream processing engine. The differences also extend to scheduling and resource management. For DataSet, tasks are scheduled from source to sink with lazy deployment of receiving tasks, which is in contrast to eager scheduling of DataStream tasks where all of the tasks are scheduled at once. This also means that resources are released in topological order when a stream ends, while in DataSet resources are released upon the termination of any task.
Final set of differences are related to state management and fault tolerance. In DataStream, snapshots of the operator states are stored periodically in memory or inside an out-of-core database. Upon a failure, the job is restarted from the last global checkpoint. In DataSet, operator

states are not stored, but intermediate results are saved to memory or file system and upon a failure, only subsequent portion of the jobs after the last intermediate result is recomputed.

3.3 A Unified Processing Model for Hybrid Computations

As can be seen in the previous section, the differences between DataSet and DataStream make creating the unified system difficult. To remove these differences, the idea is to abstract the batch data set into a window. Then, the current window model of Flink must be extended to combine the windows of batches. This section presents the extensions to the current DataStream's API so that we can incorporate the batch processing into this model.

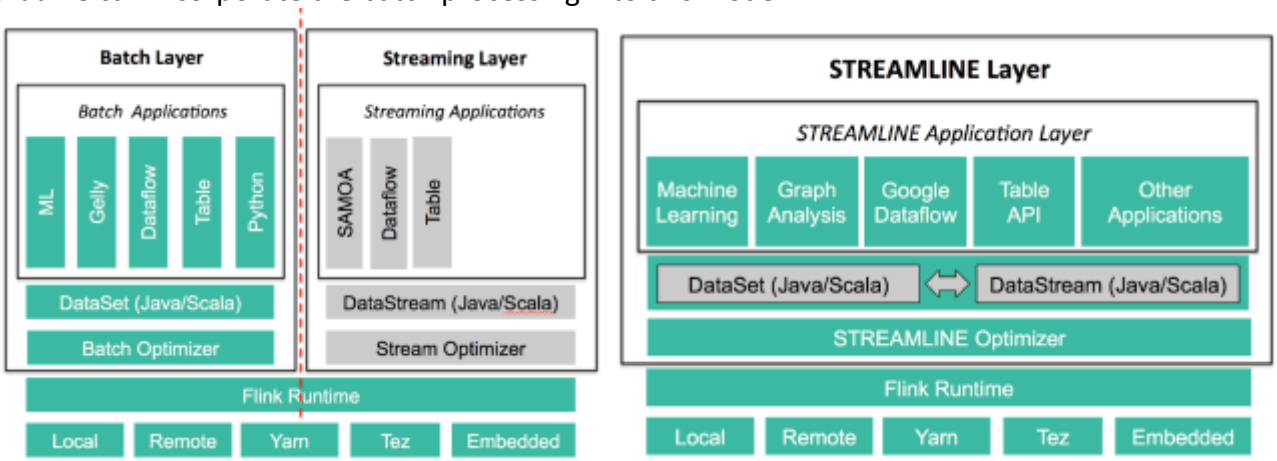


Figure 2. The current Flink system architecture (left) and the envisioned hybrid architecture (right)

At the API level, we describe the proposed operations that the DataStream should be extended with in order to support hybrid computations. Instead of having separate models for batch and stream processing, we need to design a computational model such that there is only one main data model that can represent both stream and batch data. The hybrid data processing system supports two types of inputs: batch and streaming data. Figure 2 shows the current (left) and our envisioned (right) architecture. To realize the combined computational model, we introduce side inputs. Similar to existing broadcast sets from DataSet API of Flink, side inputs are extensions to the DataStream API that enable batch datasets to be used alongside the main stream. Side inputs also enables us to implement use cases described in Section 3.1, which include but are not limited to:

- **Join of static and streaming data:** access to historical data opens up several opportunities while processing fast streaming data. It helps in filtering and enriching streaming data. Currently, only statistical summaries (such as mean, min, max, etc.) of the historical data are used to enrich the real-time data. Access to entire historical data while processing data streams allows very accurate filtering and enrichments of the real-time data. It helps with better event detection and tagging (IMR) and fast and more accurate user session analysis (IMR, Rovio and ALB).
- **Training and updating of models:** predictive analytics is the main use case of all the Streamline partners. Large machine learning models (recommender system models for Rovio and ALB, pattern recognition models for IMR) can be loaded as side inputs. As a result, these models not only are able to perform real-time predictions, but they can also be updated in real-time using the incoming streaming data. This will greatly increase the

quality and decrease the latency of the models currently being used in their production environment.

3.4 Streamline API Implementation Details

In order to integrate side inputs into Flink API, we extend the current DataStream API to include a new set of methods allowing side inputs to be loaded. Side input, itself, is a DataStream that is passed to a stream operator alongside the main streaming input. Moreover, a side input should include a materialization strategy and a partitioning scheme.

Current prototype only supports an eager materialization policy. This means, before the processing starts, the operator has to load the entire side input and only then starts processing the main stream elements. Therefore, the operator has to buffer any incoming data elements which requires additional extensions to the current DataStream operators.

The Streamline API enables multiple side inputs to be included in one operator as well. In this scenario, processing of the streaming data only begins after all the side inputs are loaded. However, depending on the task at hand and the content of the side input and the main data stream, eager materialization may not be required. We plan to explore other materialization strategies in the future.

Our current prototype includes three partitioning schemes for side input DataStream:

- **Broadcast:** the side input is replicated to every parallel instance of the stream operator. This is only suitable when the size of side input is small and it can fit in the memory of every parallel instance.
- **Forward:** this partitioning scheme is similar to one-to-one stream forwarding of Flink where shards are forwarded to next operator instance running on the same node. Here, side inputs are loaded from parallel sources (such as HDFS) and its shards are forwarded to the operator instance running on the same node.
- **Keyed:** this partitioning scheme is similar to Flink's redistributing stream where based on the given transformation each element is forwarded to a specific operator instance. In this case, keyed partitions of the main input and their associated keyed partitions of the side input are processed in the same operator instance.

Our current implementation has two drawbacks:

- **Latency**: since the side input must be loaded first, the processing of streaming elements are delayed. Time critical applications may suffer from this issue. As discussed earlier we plan to investigate the possibility of integrating other materialization strategies.
- **Fault-tolerance**: current fault-tolerance mechanism of Flink guarantees exactly-once processing of streaming elements. However, side inputs introduce new complexities that the current fault-tolerance mechanism cannot handle. In case of a node failure, both the portion of the side input that existed in the node and the streaming elements that were buffered in the node have to be restored. One possible solution is to store the buffered records in Flink state backend and simply reload the lost partition of the side input from the original source. We plan to investigate the complexity and feasibility of this method in the next prototype version.

3.5 Streamline Programming API

In order to create and bind side inputs, several new methods are added to *StreamingExecutionEnvironment* and *DataStream* classes. To lookup a side input, the Runtime Context of Flink includes a new method that returns a Java iterable collection. Listing 4 shows the new methods added to *StreamingExecutionEnvironment* and *DataStream* class and *RuntimeContext* interface.

```
public class StreamingExecutionEnvironment {
. . .
  public <TYPE> SideInput<TYPE> newBroadcastedSideInput(DataStream<TYPE> sideIn);

  public <TYPE> SideInput<TYPE> newForwardedSideInput(DataStream<TYPE> sideInput);

  public <TYPE> SideInput<TYPE> newKeyedSideInput(KeyedStream<TYPE> sideInput,
KeyTranslator kt);

. . .
}

public class DataStream <T> {
. . .
  public <TYPE, SELF extends DataStream<T>> SELF withSideInput(SideInput<TYPE> sideIn);
. . .
}

public  interface RuntimeContext {
. . .
  public <T> Iterable<T> getSideInput(SideInput<T> sideInput);

. . .
}
```

Listing 4. New methods in Flink APIs

In order to lookup a side input from *User-Defined Function*, the users must implement a Rich UDF. Within this UDF, extra information through Runtime Context can be accessed. Listing 5 shows an example of a simple application which utilizes the newly introduced side inputs.

```
DataStream<String> streamingData = env.fromElements("One", "Two", "Three", "Four");
DataStream<String> historicalData = env.fromElements("1", "2", "3");
SideInput<Integer> sideInput = env.newBroadcastedSideInput(historicalData);

streamingData.map(new RichMapFunction<>() {
            public String map(String value) throws Exception
                  {
                     Iterable<Integer> side = getRuntimeContext().getSideInput(sideInput);
                     LOG.info("Main Input: " + value + " with side input: " + side);
                     return value;
     }
}).withSideInput(sideInput);
```

Listing 5. Simple application using Streamline API

Listing 6 demonstrates the implementation of a simple recommender system use case. In this example, historical data are used to initially train a recommender system using the Matrix Factorization [16] approach. First from the existing historical data (available ratings for videos) a side input is created. Once the main input source (real-time training data) is created, both the side input and the main input source are passed to the matrix factorization model. An initial model is then trained using the historical data. This model is constantly updated whenever new training instances arrive at the system through the main streaming input. It also provides predictions in real-time whenever a prediction query arrives at the system.

```
def example(args: Array[String]): Unit = {

 val env = StreamExecutionEnvironment.getExecutionEnvironment;
 FlinkMLTools.registerFlinkMLTypes(env);


 val historicalRatings = env
   .readTextFile("hdfs://alb-cluster:8020/user/portugal-telecom/video.ratings.dataset")
   .map(line => parseSample(line));



 val streamingTrainingSet =
    env.addSource(new SocketTextStreamFunction("https://alb.com/ratings-training"))
   .map(item => parseItem(item))
   .timeWindowAll(Time.of(500, TimeUnit.MILLISECONDS))


 val streamingPredictionRequests =
    env.addSource(new SocketTextStreamFunction("https://alb.com/ratings-tests"))
   .map(item => parseItem(item)
   .timeWindowAll(Time.of(500, TimeUnit.MILLISECONDS))


 val recSys = new MatrixFactorizationModel()
   .withNumberOfFactors(40)
   .withInitUserWeights(DenseMatrix.rand(numOfUsers, n_factors))
   .withInitItemWeights(DenseMatrix.rand(numOfItems, n_factors))
   .withNumIterations(100)


 val model = recSys.fit(streamingTrainingSet, env.newForwardedSideInput(historicalRatings))


 recSys.predict(model, streamingPredictionRequests)


 env.execute()
}
```

Listing 6. A recommender system use case

The link to the repos of Side Input's source code:

https://github.com/streamline-eu/streamline-hybrid-engine

This repository will serve as the first version of unified platform for batch and stream.

Moreover, our implementation of the Side Input is inspired by the Flink Improvement Proposals 17 (FLIP-17). The Flink community as well as several members of the Streamline project were involved in the discussions that lead to the Side Input improvement plan.

Reason for the delay of pull requests of Side Input

There is a small change in the strategy for merging the source code of Streamline with Flink. We planned to create only pull requests to Apache Flink to merge our Streamline improvements into Apache Flink. However, since the time required to wait for the Flink contributors and committers to process and merge our pull requests can be a lengthy process (see Apache Flink document on Pull Request Management), it delayed the availability of our improvements to the use-case partners. By the time of this writing, the status of the proposal for Side Input on Flink website (i.e., FLIP-17) is "*Under Discussion*" which means the Flink community has not made any concrete decision about this feature yet. Therefore, we created our own fork of Apache Flink and push the improvements of Streamline to this fork. In this way, we no longer have to wait for the approval of Flink community and committers for our pull requests. The STREAMLINE fork of Flink (i.e., the STREAMLINE platform) is available here:

https://github.com/streamline-eu/streamline-hybrid-engine

4 Query Optimization Engine

In this section, we present the result of Task 1.1 Optimization: Compiler and Run-time of Work Package 1.
We present a variety of optimization techniques. Then we plan to apply these techniques to the hybrid batch-stream computation model.
To apply the optimization techniques for stream-batch hybrid computing model, we make in two steps:

- Firstly, we review the literature on optimization techniques on data stream.
- Secondly, we plan to build a dynamic optimizer that depend on the load of batch and stream , we select the most appropriate methods to apply to speed up the computation.

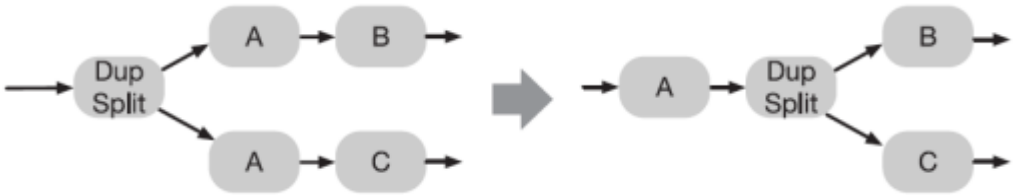
Finally, we introduce M4 [20], a technique for applying load shedding to streaming data.

4.1 Literature review on optimization techniques

Hirzel et al. present eleven techniques for stream processing optimization [13], each with different characteristic and profitability.

OPERATOR REORDERING (A.K.A. HOISTING, SINKING, ROTATION, PUSH-DOWN)
Move more selective operators upstream to filter data early.

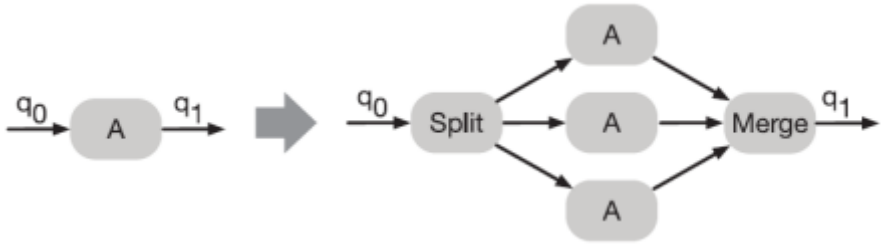
REDUNDANCY ELIMINATION (A.K.A. SUBGRAPH SHARING, MULTI-QUERY OPTIMIZATION)
Eliminate redundant computations.



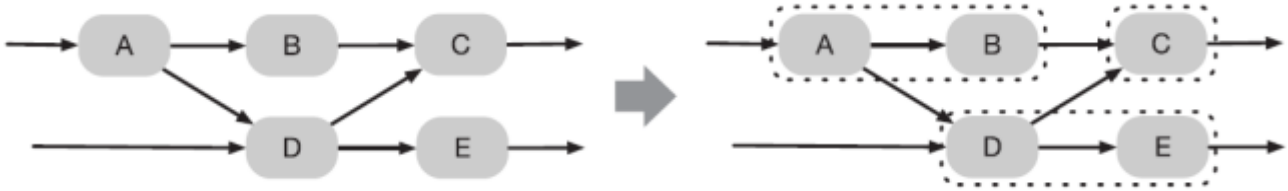
OPERATOR SEPARATION (A.K.A. DECOUPLED SOFTWARE PIPELINING)
Separate operators into smaller computational steps.

FUSION (A.K.A. SUPERBOX SCHEDULING)
Avoid the overhead of data serialization and transport.

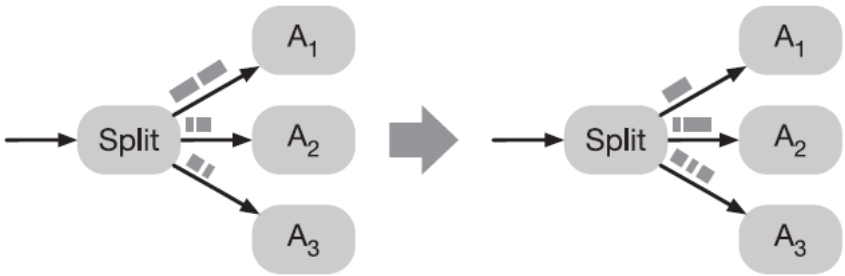
FISSION (A.K.A. PARTITIONING, DATA PARALLELISM, REPLICATION)
Parallelize computations



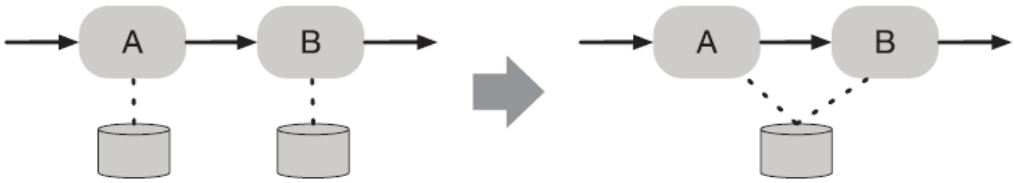
PLACEMENT (A.K.A. LAYOUT)
Assign operators to hosts and cores.



LOAD BALANCING
Distribute workload evenly across resources.



STATE SHARING (A.K.A. SYNOPSIS SHARING, DOUBLE-BUFFERING)
Optimize for space by avoiding unnecessary copies of data.



BATCHING (A.K.A. TRAIN SCHEDULING, EXECUTION SCALING)
Process multiple data items in a single batch.

ALGORITHM SELECTION (A.K.A. TRANSLATION TO PHYSICAL QUERY PLAN)
Use a faster algorithm for implementing an operator.

LOAD SHEDDING (A.K.A. ADMISSION CONTROL, GRACEFUL DEGRADATION)
Degrade gracefully when overloaded.

4.2 Potential optimization techniques applicable to batch-stream processing.

Among the eleven optimization techniques listed above, the following techniques are the potential ones that can be applied to batch-stream computation model.

Fission: we can split the data stream and batch according to the keys and then merge these results together. The problem is how to split stream-batch data efficiently to ensure the low latency and high throughput.

Load balancing: since stream-batch data is run on distributed processing system, we have to ensure that the load is divided evenly among computing nodes. This requires how we can split stream and batch data efficiently, and thus relates to the fission optimization above.

Batching: the idea of batching is somehow similar to our idea of transforming static data into big windows and streaming data into smaller windows and then process these two windows.

Algorithm selection: currently, the algorithms are designed for only stream or batch data. So we hope that new algorithms can be derived to process both stream and batch data efficiently.

Load shedding: when size or rate of the input stream and batch data is too big and exceeds the computing capacity of the platform, then we can design new load shedding technique that can remove part of the input stream-batch data to ensure the latency requirement. The problem is which data we should shed among stream and batch, and corresponding to that shedding percentage, what is the quality of service achieved at the end.

Over time, Apache Flink added supports for several of the optimization techniques above such as redundancy elimination, fusion, placement, batching, load balancing, batching, and algorithm selection. However, by the time of this writing Apache Flink still does not support operator reordering, operator separation, state sharing, and load shedding. As the first Streamline optimization, we chose the load shedding technique as the first optimization for Streamline. In the

next section, we describe how we apply load shedding to streaming data using the M4 aggregation technique [20].

4.3 Real-Time Time Series Visualization Using M4 Aggregation

In [21], we present how to use the M4 aggregation technique for real-time visualization of time series data streams. Time series data usually have high volume. Examples of such time series sources are weather data, sport analytics, and stock monitoring. The naïve way to visualize time series in real time is to transfer all the data to the user browser. However, it is usually impossible to visualize all arriving data points from a high speed data stream for the user. The possible reasons are: the displays have certain resolution and the amount of the input data is beyond processing capabilities of the browser.

As shown in [20], the amount of data which is required to plot a correct line chart depends only on the number of pixel columns and not on the amount of data. The authors derive standard SQL queries from a given plot resolution and provide a loss-free plot from only four values per pixel column which reduces the computational load of the system.

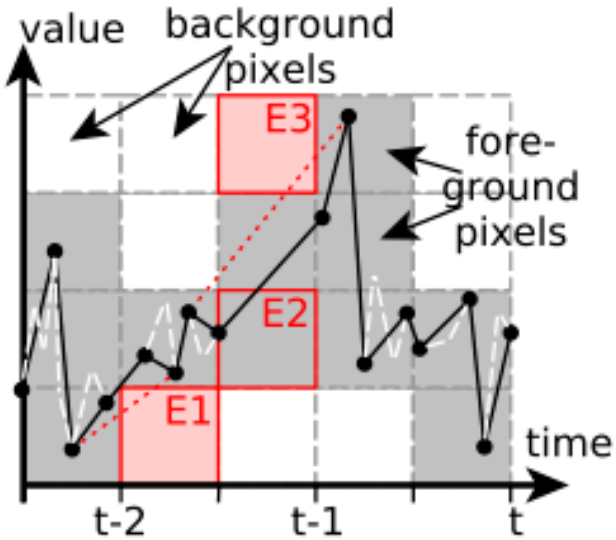


Figure 3. The M4 aggregation technique for time-series data.

Figure 3 illustrates how the M4 aggregation works. For each pixel column, M4 finds the minimum and maximum value as well as the first and the last value (minimum and maximum timestamp). All pixels which are crossed by the line connecting the extracted data points are colored and thus become foreground pixels. The intuitive approach to take only the minimum and maximum values into consideration would be insufficient. This would result in the red dotted line in Figure 1b and cause the pixel errors E1, E3 (wrongly colored) as well as E2 (not colored).

In [21], we show how the same values of the M4 aggregation technique can be computed in a parallel dataflow program, which includes both streaming and batch data sources, to allow the live visualization of incoming streaming data. Additionally, we take care of differences between event time and processing time as well as tuples arriving out-of-order, which makes processing streaming data a more complex task. We introduce I^2, an interactive development environment,

which connects distributed data analysis programs with the visualization of the results. I^2 emphasizes on two types of interactivity: (i) through code changes and (ii) through an interactive visualization GUI.

While M4 only considers finite data stored in a relational database, the real-time requirement adds several new challenges: instead of standard SQL queries, we now need parallelizable processing pipelines. Due to network delays and failures, there might be a gap between event time (the point in time a measure is taken) and processing time (the point in time the data is processed). Since data points may arrive out-of-order, we can never guarantee that the data for a pixel column is complete and possibly need to update past pixel columns in case of delayed input data. We address these challenges, as we derive a complete stream processing pipeline from a given plot resolution and the length of the depicted history as shown in Figure 4. The pipeline mainly consists of three steps each of which can be executed as an operator with possibly multiple parallel instances:

- **Watermarks**. Watermarks flow through the pipeline alongside the regular data and propagate the progress of event time. A watermark of time t_w means that no later processed event will have a timestamp $t_e < t_w$. We input watermarks at the data source of our pipeline to mark the smallest timestamp which is still covered by the live plot. Hence, we update pixel columns in case data arrives out-of-order. However, we avoid unnecessary processing of out-of-order data which arrives so late that the corresponding pixel column of the live chart is no longer displayed.
- **Windowing**. We apply a time window function which splits the stream into finite data chunks spanning the time of one pixel column. We then compute the M4 aggregates over these windows and respectively for each pixel column.
- **Value compression**. Finally, we map the results of the aggregation to the value space of the y-axis which allows us to represent each value with less bytes.

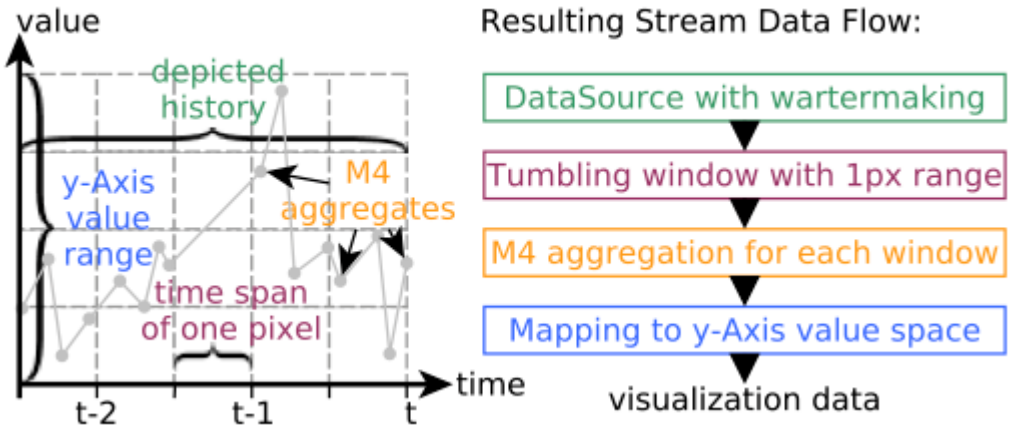


Figure 4. Deriving a stream data flow program for the real-time visualization of time-series data with M4.

The approach used in I^2 for interactive visualization of high speed data stream is a load shedding technique, i.e., it performs early data reduction. Using a parallelized version of the M4 aggregation

technique, I^2 reduces the number of elements which need to be processed in a combined streaming and batch processing workload. Therefore, it can reduce latency and increase throughput.

The relevant publications and source code of M4 technique for visualization can be found in this repositoryhttps://github.com/streamline-eu/i2. The results of this work were published in EDBT 2017 demo paper session and received a best demo paper award [21]:
Jonas Traub, Nikolaas Steenbergen, Philipp Grulich, Tilmann Rabl, Volker Markl. I2: Interactive Real-Time Visualization for Streaming Data. EDBT 2017: 526-529.

5 Fault Tolerance

In this section, we present our results for Task 1.2 Fault Tolerance: Managing Stateful Streaming of Work Package 1. We first provide an overview of types of fault tolerance. Then, we discuss the requirements of fault tolerance for hybrid computation.
In the distributed stream processing system, the failure of one node can significantly affect overall processing if there is no appropriate fault tolerance technique to prevent failures. Such failure can cause the loss of large amount of processed data, making the final result incorrect. The situation is similar for the hybrid processing system. Therefore we must incorporate a suitable and efficient high-availability mechanism to allow seamless processing of stream and batch data in spite of hardware failure. In this section, we describe the fault tolerance technique used in the unified system for data at rest and data in motion. This section includes two main contents:

- Firstly, we list the three types of fault tolerance used in the literature.
- Secondly, we point out the requirement for fault tolerance technique in the hybrid system and then choose the appropriate technique for this system.

5.1 Types of fault tolerance

This section lists three methods to achieve rollback recovery, with each having different characteristics of redundant computation, checkpoint technique, and logging at other node. Therefore, each method is suitable for a specific scenario. We describe its properties and select the most appropriate one to extend to satisfy the requirement for fault tolerance in hybrid system.

The first method in the rollback recovery is passive standby. In this method, each primary node sends the changes of its state to the secondary node. Each checkpoint message captures the changes to the states of input queues, operators, and output queues since the last checkpoint message was completed. If a primary node is down, the secondary takes over and sends all tuples from its output queues to the downstream nodes. This method incur high runtime overhead proportionally to the checkpoint duration and the size of the checkpoint messages.

The second method is active standby. Each secondary node simultaneously receives tuples from upstream and processes them in parallel with the primary. In case of failure-free, the secondary does not send any output tuples downstream, but only logs in its output queues. Since this method process duplicated tuples, the computing resource is double.

The final method is upstream backup in which upstream nodes play the roles of backups for their downstream neighbors by logging tuples in their output queues until all downstream neighbors do not need these tuples anymore. Determining the maximum set of logged tuples that can be released is crucial in this method. In case of failure, this method has to rebuild the primary's state from an empty state in the secondary node. So it incurs high recovery cost.

Among the three types of fault tolerance mentioned above for rollback recovery, passive standby is the most appropriate method for our hybrid system to base on. The first reason is that active standby requires significant computing resource to replicate the computation for ensuring fault

tolerance. The second reason is that upstream backup has long recovery time. In the next section, we sketch the requirement that the proposed fault tolerance must satisfy.

5.2 Requirement for fault tolerance in hybrid system

The first requirement for the fault tolerance technique used in the hybrid system is that it does not require too much computing resource for fault tolerance. The computing resource must be saved mostly for actual computing of result. So, we will not duplicate the tuples as in the case of active standby.

To use the passive standby method efficiently, it requires a way to reduce the state size of operators to reduce the checkpointing cost as well as the recovery cost. This is because the state size directly affect the time to checkpoint and recover. So, smaller state size can achieve smaller checkpoint overhead and faster recovery.

Transaction processing requires *consistency guarantees*. To achieve one-pass execution, we envision our fault tolerance method to provide novel feature of consistent snapshot generation. This requires the inputs, outputs and intermediate operator output states to be saved for recovery.

6 Incremental Computation

In this section, we present our results for Task 1.4 Incremental Computation of Work Package 1. This section includes two main contents:

- Firstly, we define the user-defined windows (UDW) concepts which is necessary for incremental aggregation. Then, we classify two kinds of UDWs.
- Secondly, we propose the Cutty framework that support strategy to incrementally pre-compute higher-level aggregates [14].

6.1 User-defined Windows

Traditional aggregation technique of periodic windows have been extensively studied in the past through the use of aggregate sharing techniques such as Panes and Pairs, no work has been put in optimizing the aggregation of non-periodic windows. This section gives an example of such non-periodic window and classify it into two kinds of windows: deterministic and non-deterministic.

Consider the monitoring application in stock market that continuously receives records representing stock trades. Each record contains a volume (how many units were traded) and a price, per traded unit. A stock trader wants to see the volume-weighted average price of a stock over the last 10 minutes, reported every 5 minutes. However, when the stock price falls below a certain threshold (e.g., the trader can buy the stock in a low price), the trader wants to receive an update every 2 minutes, with a weighted price average of the last 5 minutes. As specified by the trader, when the price falls below $10 on the 25th minute, the slide becomes more frequent (every 2min) and the window range becomes shorter (5min). More formally, the window definition goes as follows:

$$\text{window} = \begin{cases} \texttt{SLIDE=5min; RANGE=10min } if \text{ price} > \$10 \\ \texttt{SLIDE=2min; RANGE=5min } if \text{ price} \leq \$10 \end{cases}$$

This user-defined window (UDW) dynamically changes its range and slide according to the incoming stream. The semantics of such windows have not been defined or supported by systems in the past. By using this UDW, we propose the sharing strategy in which Incremental aggregation is the core functionality of it.

Classifications of UDWs. We distinguish two classes of UDWs, namely deterministic and non-deterministic. In short, deterministic windows are the one for which we can apply slicing efficiently as described previously, while non-deterministic are all the rest for which we cannot. Deterministic windows can be declared with a discretization function:

$$\text{Discretize} : f_{disc} \times Seq(T) \rightarrow Seq(Str(T))$$

Intuitively a window function is deterministic if at the time that it processes a record, it can decide whether that records marks i) the beginning of a window or ii) the end of a window. For instance, a periodic count window of fixed range and slide is deterministic, since when a record arrives, an

internal counter can affirm whether a new window begins with that record. Similarly, a punctuation window is deterministic, since (by definition) a punctuation marks the beginning of a window.

Periodic windows are supported by most existing event processing systems, and are trivially subsumed by deterministic. Tumbling windows are periodic and thus, deterministic. Session and snapshot windows by definition begin with the first record of the session that marks the window's beginning and end a timeout record which is injected in the system (e.g., a watermark). Moreover, lower-bound landmark windows begin from a given landmark record, that marks the beginning of a record and end upon a punctuation or a predefined length.

Intuitively, non-deterministic windows cannot declare immediately whether a record begins a window or not, i.e., they need to examine more records in order to take such a decision. As a result, slicing for non-deterministic windows is not possible and we have to fall back to best-effort techniques. A typical example of a window that is non-deterministic is a window which every 5 seconds outputs the last 10 records of the stream. If we assume that a record r1, arrives during the first second of a window and the next second, another 10 records arrive, r1 will not be part of the next window. Thus, if the rate of the stream cannot be known in advance, such a window cannot be deterministic; the window function cannot specify whether a record is going to be part of the next window at the very moment of the record's arrival. In this work, we focus on shared aggregation of deterministic windows, for which slicing is applicable.

6.2 CUTTY framework for incremental aggregation

In this section, we describe a general framework for aggregating multiple overlapping windows. This aggregation framework exploits the properties of deterministic windows using Cutty, a novel pre-aggregation technique. Furthermore, it utilizes higher-order pre-aggregated partials which adds sharing capabilities for deterministic windows.

A Thorough Example. Figure 5 depicts a full example of the execution of Cutty for aggregating a set of deterministic UDWs. As the set of the deterministic UDWs dictate (f_{disc}'s at the top), partial aggregation is applied incrementally only within the intervals that mark the beginning of windows (dashed vertical lines). At the beginning of every interval, the active partial resets to the initial value 1A and maintains the current pre-aggregate until it forms a complete atomic slice (in this example, (P(s[1; 2]); P(s[3; 5]); P(s[6; 7]); P(s[8; 8])). Before the active partial resets, its value is stored for further reuse (central rectangle in the figure). When a window ends we have everything to compute the full aggregation from the partials. For instance, in the case of f_a(s[3; 8]), upon getting notified at the consumption of s9 that window s[3; 8] is complete we are ready to compute the full aggregation. We can derive the full window aggregation by re-using all precomputed slices P(s[3; 5]); P(s[6; 7]) and P(s[8,8]) which are already stored.

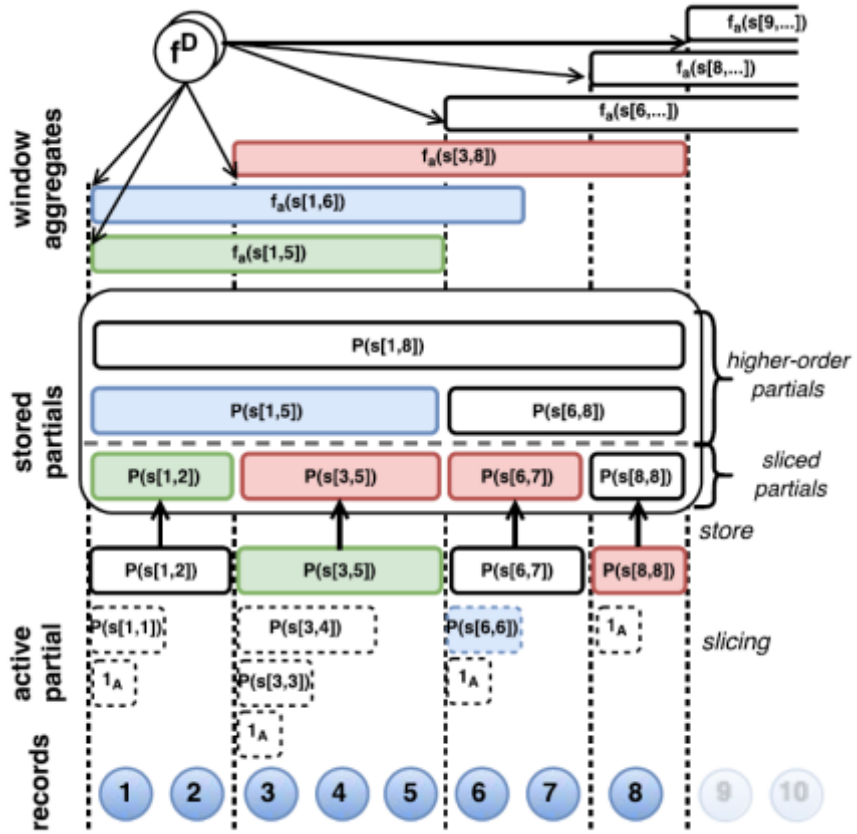


Figure 5. An example of Cutty on deterministic windows

Sharing Higher-Order Partials. From the example in Figure 5, we observe that, simply sharing sliced partial aggregates does not eliminate redundancy when computing full window aggregations. For example, with a typical lazy evaluation slices P(s[1; 2]) and P(s[3; 5]) would have to be combined twice: once for computing fa(s[1; 5]) itself, and once for computing fa(s[1; 6]). Instead, if f_a(s[1; 5]) was stored, f_a(s[1; 6]) could be computed by f_a(s[1; 5]) together with the current partial. In principle, the higher the number of overlapping windows, the more reduce calls are repeated for each window merging operation. To deal with this issue we can exploit an eager pre-aggregation strategy to incrementally pre-compute higher-level aggregates. Eager pre-aggregation has been used previously to support aggregate look-ups on data streams (e.g., B-Int[19], FlatFAT [17]) at the cost of additional space and update computation requirements. It has been shown, however, that these costs provide a strong trade-off when eager aggregation is applied even in a per-record granularity of a data stream [18, 9].

In our case, we apply this technique for enriching atomic sliced partials with higher order reusable partials, thus, effectively reducing aggregation cost at a low additional memory footprint. Furthermore, eager aggregation can be used as a best-effort fallback solution for non-deterministic UDWs, where slicing cannot be applied. The key idea is that we can eagerly evaluate

higher-order aggregates and maintain them in efficient data structures in order to support arbitrary lookups, described in more detailed in the next section.

The main component that executes the aggregation is the Cutty aggregator which consumes the enriched stream and pre-aggregates reactively. In the next section, we present the discretizer which injects windowing information into the data stream and then proceed with presenting our shared aggregator.

Shared Discretization. The shared discretizer is able to multiplex multiple UDWs, coming from multiple queries. The basic functionality of the discretizer is to consult all the UDWs for each incoming record, enrich it with the needed information and pass it downstream to the aggregator.

Deterministic UDWs. As we have seen in Section 3.2, deterministic functions allow us to know whether a record begins or ends a window, exactly at the moment of the record's arrival. This gives the discretizer the ability to give hints to the aggregator, alleviating the aggregation process from the need for window management and bookkeeping. Hence, the aggregator simply operates on a marked stream, without any knowledge about the specifics of the UDWs.

The discretizer associates a set of window begin/end identifiers with each incoming record. These identifiers are used by the aggregator to apply slicing incrementally. Consider for example the shared discretization of windows produced by two UDWs as shown in the example in Figure 3. The upper UDW has a range of 4 and slide 2 while the one in the bottom has a range of 5 and slide 3. The dashed vertical lines mark the begin of each individual window, as indicated by the UDWs. The discretizer consults the UDWs and injects window begin" markers in the stream (as depicted on the right). For instance, on record 0, there are two window begins, namely windows 1 and 2. Similarly, the discretizer consults the UDWs and enriches the stream with window end"markers. In our example, window 1 ends with record 3 (and indicated by the UDW upon processing record 4).

Non-Deterministic UDWs. Non-deterministic windows cannot indicate whether the current record of the stream begins a new window. To this end, non-deterministic UDWs have to indicate possibly expired records that should be removed from the head of the current window, and notify when the window has to be emitted. Since the shared discretizer operates on multiple UDWs at the same time, it has to i) derive the records that expire across all UDWs i.e., the intersection of records that all UDWs have declared as expired ii) store and track the beginning of each active window. Non-deterministic windows in our aggregator architecture are handled by the underlying aggregate store as described in [17]. For the lack of space, we will omit the details of how expired records are handled by our aggregator, and refer the reader to the original work.

Shared Aggregation. Efficient Aggregate Storage. The aggregator needs to maintain partials in memory and retrieve them efficiently. To this end, we designed an aggregate store that provides support for range queries over the aggregates (e.g., when multiple partials have to be combined for a window emission). The store supports three basic operations:

- append(partial_id, partial): Adds a partial at the end of the store where partial_id is an identifier for the provided partial.

- merge(from, to): Computes result of P(s[from; to]). Most of the time the aggregator is interested in looking up a full aggregation starting by from. In that case, we will use the shorthand call merge(from).
- removeUpTo(partial_id): Removes all given partials from the store up to partial_id.

We considered two evaluation strategies for the store, a lazy using a circular xed-sized array, and an eager which builds on FlatFAT [17] a pre-allocated memory circular heap-based data structure. For the rest of this section it should be assumed that the store follows an eager strategy on a binary tree unless stated otherwise. The complexity of storage and retrieval plays a very important role in the performance of our aggregation technique and is analyzed in Section 5.4.

Aggregating Enriched Streams. The functionality of the Cutty aggregator is summarized in Algorithm 1. The aggregator maintains a single active partial on which it applies incremental aggregation per record arrival. Effectively this is an execution of stream slicing, where each slice spreads between records that mark consecutive window begins. In case a new window begins (which occurs when the set WIDbegin is not empty) the aggregator has to start a new partial pre-aggregation. In that case, the current active partial is stored in the aggregates store and the active partial resets back to its initial value (1A). In any other case the aggregator simply applies a single combine operation to update its active partial. Mind that the aggregator keeps track of the first record of every active window since it has to be aware of the specific range to aggregate when a window ends. For every window that ends in W_{end}, the aggregator combines its active partial with the stored partial of the interval that starts at the beginning of the window.

Algorithm 1 Agg $\langle partial, 1_A, \oplus, \texttt{lift}, \texttt{lower}, store, begins \rangle$

```
1:  upon event ⟨ r_i, WID_begin, WID_end⟩ do
2:      if WID_begin ≠ ∅ then
3:          store.append(i, partial)
4:          partial := 1_A //reset partial
5:           for each  w ∈ WID_begin
6:              begins[w] = i //mark begin for w
7:       for each  w ∈ WID_end
8:          start := begins[w] //retrieve begin of w
9:          begins.remove(w)
10:         store.removeUpTo(min(begins)) //gc
11:         emit ⟨w | lower(store.merge(start) ⊕ partial) ⟩;
12:     partial := partial ⊕ lift(r_i) //online aggregation
13: end
```

Storage Costs. A very important feature of Cutty aggregation for deterministic UDWs is that it generates a minimal amount of sliced partials needed for any shared window computation. In the worst case, Cutty stores only as many partials as the number of active windows, compared to other slicing techniques [18] that generate twice as many partials. The main idea lies at the

observation that only windows that end at a specified record r_i would need to aggregate up to i. Since no other windows would ever need to start or end at this index later, incremental aggregation can continue until it reaches a record which starts a new window.

For non-deterministic window aggregations, it is not possible to apply slicing due to the limited knowledge of the active windows. However, in that case Cutty utilizes the store, thus, bounding its performance to the current state-of-the-art approach [17]. Expired record removals and full window aggregates are executed based on the discretizer-injected information. Partial aggregation sharing is therefore achieved only via the eager strategy of the aggregate store.
The Cutty aggregation was published in CIKM 2016: P. Carbone et al. . Cutty: Aggregate Sharing for User-Defined Windows. CIKM 2016.

7 Conclusion

In this document, we have outlined our plan for seamless integration of data at rest and data in motion. The design presented enables the joint processing of both DataSet and DataStream by introduction of side inputs into the DataStream API. This essentially allows a static dataset to be loaded into a streaming operator and be used alongside a streaming source. To realize this plan, new components as well as modification to existing Apache Flink's underlying engine has to be made. The requirements for a hybrid processing system can be divided into five main categories: new computation model, novel hybrid operators, state management and fault tolerance, optimization techniques and incremental computation.

The new computation model relies on side inputs as the link between static data and streaming data. One benefit of the side input is that it is a mere extension of the current DataStream API. This means that any changes made to the current Flink APIs are backward compatible.
The new unified processing platform allows implementation of use cases that were previously not feasible. As outlined in section 3, Streamline partners can use the new API to implement several different use cases such as video recommender systems, event detection and tagging and user session analysis.

To make the transition from single execution environment to hybrid seamless for the programmers, the new architecture is to be built on top of the existing streaming execution environment. This means that any of the existing streaming operators should be usable within the new hybrid architecture. It should also support defining operations on an easy and programmer-friendly API. To achieve this, some changes have to be made to the existing Apache Flink engine, which are outlined in Section 3.3 and Section 3.4.

We introduced a new load shedding optimization based on M4 aggregation which can be used to reduce data stream early on. We show how the M4 aggregation technique can be computed in a hybrid parallel dataflow program (i.e., data flows with both streaming and batch data sources). We introduce I^2 interactive development environment which exploits parallel version of M4 to process rapid data streams in real time for interactive visualization.

Incremental computation is required to speed up the processing of evolving data streams. To that end, we introduced CUTTY framework, a novel pre-aggregation technique that utilizes higher order pre-aggregated partials that adds sharing capabilities for certain type of windows.

Initially, we planned to merge Streamline improvements into Apache Flink using Pull Requests (PRs). However, since the pull request management process of Flink is quite lengthy, we have decided to push Streamline improvements to a fork of Flink.

References

[1] T. White. Hadoop: The definitive guide. " O'Reilly Media, Inc.", 2012.
[2] J. Dean, and S. Ghemawat. "MapReduce: simplified data processing on large clusters." Communications of the ACM 51.1 (2008): 107-113.
[3] A. Alexander, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser and V. Markl, "The stratosphere platform for big data analytics," The VLDB Journal, vol. 23, no. 6, pp. 939-964, 2014.
[4] The Apache Software Foundation, "Apache Flink," 31 12 2015. [Online]. Available: flink.apache.org. [Accessed 17 May 2016].
[5] S. Ewen, K. Tzoumas, M. Kaufmann and V. Markl, "Spinning fast iterative data flows," Proceedings of the VLDB Endowment, vol. 5, no. 11, pp. 1268-1279, 2012.
[6] O. Boykin, et al. "Summingbird: A framework for integrating batch and online mapreduce computations." Proceedings of the VLDB Endowment 7.13 (2014): 1441-1451.
[7] J. Meehan, et al. "S-Store: streaming meets transaction processing." Proceedings of the VLDB Endowment 8.13 (2015): 2134-2145.
[8] R. Kallman, et al. "H-store: a high-performance, distributed main memory transaction processing system." Proceedings of the VLDB Endowment 1.2 (2008): 1496-1499.
[9] R. Casado, “Lambdoop, a framework for easy development of Big Data applications,” in NoSQL Matters Barcelona, 2013.
[10] Apache Kafka project.
[11] K. Shvachko, H. Kuang, S. Radia and R. Chansler, "The hadoop distributed file system," in IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), 2010.
[12] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans and E. Baldeschwieler, "Apache hadoop yarn: Yet another resource negotiator," in Proceedings of the 4th annual Symposium on Cloud Computing, 2013.
[13] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm. 2014. A catalog of stream processing optimizations. *ACM Comput. Surv.* 46, 4, Article 46 (March 2014).
[14] P. Carbone et al. Cutty: Aggregate Sharing for User-Defined Windows. CIKM 2016.
[15] M. Zaharia, et al. "Spark: cluster computing with working sets." *HotCloud*10 (2010): 10-10.
[16] Y. Koren, R. Bell, and C. Volinsky. "Matrix factorization techniques for recommender systems." *Computer* 42.8 (2009): 30-37.
[17] Tangwongsan, K., Hirzel, M., Schneider, S., & Wu, K. L. (2015). General incremental sliding-window aggregation. *Proceedings of the VLDB Endowment*, *8*(7), 702-713.
[18] Krishnamurthy, S., Wu, C., & Franklin, M. (2006, June). On-the-fly sharing for streamed aggregation. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data* (pp. 623-634). ACM.
[19] Arasu, A., & Widom, J. (2004, August). Resource sharing in continuous sliding-window aggregates. In Proceedings of the Thirtieth international conference on Very large data bases-Volume 30 (pp. 336-347). VLDB Endowment.
[20] Uwe Jugel and Volker Markl. M4: a visualization-oriented time series data aggregation. Proceedings of the VLDB Endowment, 2014.
[21] Jonas Traub, Nikolaas Steenbergen, Philipp Grulich, Tilmann Rabl, Volker Markl. I2: Interactive Real-Time Visualization for Streaming Data. EDBT 2017: 526-529.