Array-Based Stream Processing in Apache Flink

Jaka Mohorko

Master of Science
Computer Science
School of Informatics
University of Edinburgh
2020

Abstract

As increasing amounts of data are generated, real-time processing of high-volume data is becoming a necessity. To address that need, several stream processing engines (SPEs) were developed. These SPEs are primarily used for SQL-style relational operations on grouped streams. Applications in fields such as the Internet of Things often require array-based operations on streams, which are not supported by traditional SPEs. To bridge the gap between grouped stream processing and array-based operations, SPEs are often loosely coupled with numerical frameworks such as MATLAB or R. Such loose coupling of system, however, incurs large communication costs and increases implementation complexity.

In this project, we create and evaluate an array-based processing framework extension to the Apache Flink SPE. This allows for complex workflows involving relational queries, as well as array-based algorithms to be performed in a single system, using a unified query language. We build upon the Flink framework, retaining the base relational query functionalities while providing an array-function interface where users can define custom array-based operations without worrying about the underlying data structure. We achieve significantly better performance as compared to using a loosely-coupled Matlab integration in Flink and achieve competitive performance compared to native Flink operators.

Acknowledgements

I would like to thank my supervisor, Dr. Milos Nikolic, who guided me through the project and offered support when the work seemed overwhelming. Advice on keeping the project streamlined and what parts to focus on made my completion of all my goals for this project possible.

I would also like to thank my friends who supported me during my moments of panic as I was writing. Without your support I would likely still be worrying about the completion of this project, as opposed to completing it.

Table of Contents

Chapter 1

Introduction

Real-time data processing has become an emerging topic in the world as increasingly larger amounts of data are being generated. Internet of Things (IoT) devices, sensor networks, location-tracking, as well as many other fields have made real-time, high-volume data processing a necessity. To address that need, stream processing engines (SPEs) provide infrastructures supporting analysis and operations on potentially unbounded streams of real-time data [10, 11, 31, 12, 1]. These SPEs are primarily developed for relational operations on streams, such as grouping by stream source, filtering and joining streams based on a set of conditions.

While SPEs often support custom operations on stream data subsets in the form of "windows", where streams are split into finite buckets, they provide no guarantees of data uniformity and consistency and do not check for missing data. Data driven processing often requires complex workflows including various domain-specific algorithms, such as the Fast Fourier Transform (FFT). For efficiency purposes, these algorithms assume that data is in an array format and achieve sub-par performance otherwise. Furthermore, they often require uniform arrays of a specific size with equally spaced data as inputs, which cannot be reliably created through conventional SPEs.

As such, numeric computation environments such as R or MATLAB, which offer efficient computation methods of said domain-specific operations, are used to process streaming data. However, these environments lack the support for relational stream operations supported by SPEs. The need for both functionalities in fields such as machine learning, IoT sensor data processing, or digital signal processing (DSP) led to loose coupling of computation environments within stream processing engines, or vice-versa. Such coupling of systems, however, often requires a deep understanding of SPE frameworks, making implementation very difficult. Furthermore, while com-

putation in the aforementioned numerical environments is efficient, the efficiency is overshadowed by high communication costs incurred when transferring data between them and SPEs, leading to poor performance.

Implementations such as TrillDSP [23] or WaveScope [19] are examples of systems with tight integration of DSP specific tasks within SPEs. Employing techniques such as data resampling and interpolation, they expose input streaming data in the form of arrays representing signals to users. TrillDSP and WaveScope provide query languages consisting of operations familiar to DSP experts within SPEs, allowing for deep integration of analysis and processing tasks in a single system.

Exposing streams in uniformly-sampled sets, however, is a requirement in many areas other than DSP. Researchers in database management systems (DBMS) have recognized the need for support of efficient array-based operations IoT, or image processing [14], resulting in the creation of several array DBMS. Their findings showed that simulating arrays on top of relational models was inefficient and proposed implementation of array-based databases, as arrays were the most commonly used data model in the aforementioned fields [25].

To address the low performance and complexity of loosely-coupled systems in stream processing and the need for array-based operations on streams, we create a tightly-coupled framework in the Apache Flink SPE [10] that allows for array-based processing. Flink is a widespread open-source SPE, favoured by a large user-base and has a large amount of contributors providing a wide array of custom plugins and libraries. For these reasons, as well as its competitive performance [13], we chose Flink as the platform for our system.

The primary focus of our system is to assemble stream input events into arrays, which are exposed to users within an array-function interface, abstracting the tempo-relational component of data. This allows users to perform array-based operations on streams without needing to worry about underlying data structure consistency and uniformity. To expose streams in array form, we build upon techniques used in TrillDSP [23]. We use event timestamps to index into arrays, re-align timestamps to match index intervals and interpolate missing data when array indices are not populated.

Exposing an array-function framework within Flink removes the need for coupling with outside numerical environments, reducing the performance overhead and implementation complexity. It allows for complex workflows to be expressed within a single system, using one query language for all workflow components. Our system supports integration of its array module with other native Flink queries, combining the relational

aspect of SPEs and the array-processing aspect of MATLAB/R in one package.

Furthermore, we provide support for incremental computation on streams, where streams are partitioned into partially overlapping windows. Various analysis techniques, such as overlap-add [16] used in DSP, require computation over overlapping data partitions. When there is significant overlap between stream partitions, elements can be evaluated several times, causing large amounts of redundant computations to be performed. However, by allowing output forwarding between stream window evaluations, we allow for incremental computation methods, such as the Sliding DFT [21], thus alleviating the performance cost of overlapping data computation.

Within our system we provide two sample use-case implementations showcasing the use of our system for DSP tasks and data encryption. Furthermore, we give users access to an interface exposing data streams as arrays to users, allowing for creation of any custom domain-specific operation that requires arrays as input.

1.1 Project Contribution

The contributions of this work are:

- The design and implementation of an Apache Flink framework supporting general-purpose array transformations and operations on real-time streaming data without the need for integration with external tools. We create the framework through uniform array creation on top of non-uniform data streams through resampling and interpolation.
- Full integration with native Flink operators for both grouped and non-grouped stream processing.
- An array-based function interface implementation abstracting the notion of time and any back-end operations. The interface provides support for use cases for encryption and spectrum analysis, as well as allows for the implementation of custom array functions.
- An interface for performing incremental computation on sliding windows where users have full control over data forwarding and computation methods.
- The performance evaluation and benchmarking of our implementation compared to loosely integrated systems and native Flink operations.

1.2 Chapter Outline

Chapter 2: Background and Related Work first discusses the architecture and components of the Apache Flink stream processing engine, detailing components used in our system. It then presents background information on array-based processing and work related to our system, outlining the similarities and differences.

Chapter 3: Design describes the requirements of our system and how its functional parts relate to said requirements.

Chapter 4: Implementation describes how the design was implemented within the Apache Flink framework. Furthermore, it details the challenges faced during implementation, as well as the limitations of our system.

Chapter 5: Evaluation presents the evaluation results of our system where it was compared to loosely-coupled array function implementations using MATLAB and estimates the overhead of our system compared to native Flink stream operations.

Chapter 6: Conclusion contains the concluding thoughts, outlines future work and summarizes the project results.

Chapter 2

Background and Related Work

In this chapter, we start by covering the basic concepts and components of Apache Flink [10] relevant to our system. We continue to address what array-based processing is and how the concept translates to our work. We also highlight some systems and fields such as Array Database Management Systems where arrays are used as the primary data structure. In the last section, we describe related work on the topic of stream processing and signal stream processing which incorporates similar techniques for stream-to-signal conversion as our system for array creation. Finally, we contrast some relevant incremental computation techniques on sliding windows to our framework for incremental computation.

2.1 Apache Flink

Apache Flink [10] is an open-source project that provides an interface for implementing and executing operations on bounded or unbounded data streams. Flink provides two core APIs, the DataSet API for bounded data set processing and the DataStream API for unbounded streams, which enables stream processing within Flink. For this project, we focus on extending the DataStream API with our system.

The core Flink architecture is comprised of three process types: client, job manager and task manager. Clients transform program code into dataflow graphs described in Section 2.1.1, while the job manager coordinates the dataflow execution. Lastly, the task managers execute Flink operators (Section 2.1.2), which produce output streams. In our project, we make use of the underlying architecture for dataflow coordination, state management and scheduling, and create custom operators which enable array-based processing in Flink.

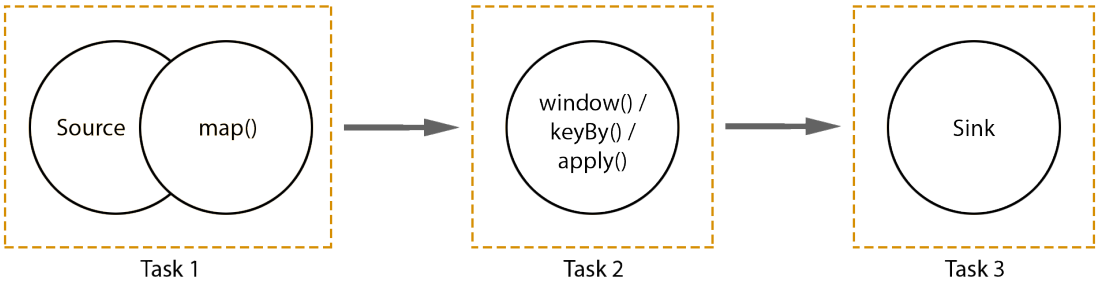

Figure 2.1: Operator chain in Flink showing groupings of operators into tasks.

2.1.1 Dataflow Graphs

The Dataflow Graphs in Flink correspond to the flow of streams between operators. The graph in Figure 2.1 shows how operators are grouped into tasks each associated with their input and output data stream. Data is pipelined from one task to another, with each task executed by one thread. Operators are grouped into tasks to reduce the overhead caused by thread communication [6]. At the end of a dataflow sequence, a sink can be specified which collects and outputs the processed stream data using the user's choice of output method.

2.1.2 Operators

Operators are used to transform data streams. They apply native or user-defined functions to the input data and output a transformed data stream. Operators represent nodes of a Flink dataflow graph and can be chained together to create advanced execution graphs. Flink defines a large amount of native operators, providing various modes of aggregations, joins between streams and other methods, as well as providing operators that allow user-defined functions to be executed.

Operators are the Flink construct where streams are modified and exposed to users. Our system implementation is based on creating operators that modify the streams and allow for array creation and modification.

2.1.3 Time Model

Flink mainly associates stream events with two different measures of time: processing-time and event-time [6]. Processing-time denotes the current system time of a machine running the Flink operator when an event is being processed, while event-time refers to the time at which the event was created at the source. Within our system we make

use of event-time timestamps, using them to measure the distance between elements to facilitate the creation of a uniformly sampled stream.

Many core principles described by the Google Dataflow Model [4] are adopted by the Flink framework, including the use of watermarks to measure global progress. Watermarks are used to track pending events in the Flink distributed system, allowing for distinction between missing and late events [3]. They tell the system that there are no more events with a time lower than the watermark time waiting to enter an operator.

Watermarks are mostly created at the origin stream and propagated through the system by operators, however, the Flink framework provides a `TimestampedCollector` class which can be used to emit events with new timestamps onto the output stream. As our system relies on realignment and interpolation of missing values to create uniform streams, event watermarks must be modified and emitted in correct order to maintain a watermarking procedure consistent with that of the Flink runtime engine.

2.1.3.1 Windowing

To perform computation on unbounded streams, said streams are most often split into `Windows`, which partition a stream into finite segments. Windows have stream events assigned to them by `Window Assigners`, which determine the rules on what event is slotted into what window. Flink provides different window assigners [6]:

- `Tumbling Window` - Creates fixed-sized windows that do not overlap
- `Sliding Window` - Creates fixed-sized windows that include an additional *sliding* parameter, which determines a time-window after which a window is evaluated. This can lead to potential overlap between windows if the sliding parameter is smaller than the window size.
- `Session Window` - Groups elements by sessions of activity where nearby elements are slotted into the same window, and windows close after a specified period of inactivity.
- `Global Window` - Assigns all elements to a single window. Most often used in conjunction with a certain `trigger` which specifies when the window should be evaluated, at which point elements are likely evicted from the global window.

Within our system's workflow, we use global windows with a counting trigger, which evaluates the window contents when a specified amount of elements has been

received. As our system achieves a unique property where events are sampled uniformly, counting events is equivalent to specifying a window time-frame, where the time-frame is defined by $windowElementCount * samplingInterval$.

2.1.4 Stateful Processing

Flink supports "stateful stream processing" through managed states. While Flink operators that apply functions to input streams mostly process each event and produce a corresponding output, some operations, such as aggregation or updating machine learning models require operators to keep track of past results and data [6]. Therefore, each operator can create and manage its state, maintaining and updating it with data persisting across multiple function invocations over input events.

There are two different varieties of managed state within Flink - `Keyed State` and `Operator State`. The `Operator State` is a state associated with a single operator, and is declared separately for each parallel task instance [9]. It is mostly used when a stream cannot be partitioned into groups via a "key". For the operator implementations within our system, only the `Keyed State` is needed.

2.1.4.1 Keyed State

Input streams within Flink can be partitioned by an user-specified key, resulting in a `Keyed Stream`. When operating on a `Keyed Stream`, Flink manages state separately for every key instance within the stream, providing each operator with an appropriate `Keyed State` on execution. Several types of `Keyed State` are present within Flink, all exposing a different API [9]. Within our system we make use two state types:

- `ListState` - an append-only state which maintains a list of values per key and supports an `add` operation.
- `ValueState` - a single-value state which supports an `update` operation allowing mutations to the value stored within the state.

2.2 Array-Based Processing

The array data structure lies at the core of many algorithms used in fields such as IoT or DSP. Array Database Management Systems (DBMS) are an example where the need for an underlying array structure was identified, following study results showing that

layering arrays over table structures results in poor performance [25]. SciDB [8] and MonetDB [20] are examples of DBMS that provide column-oriented databases, making use of arrays as the underlying structure to facilitate efficient array computation.

Within this project we define array-based processing as operations on arrays, producing transformed arrays as output. Arrays are used to represent uniformly sampled data, consisting of events at set time intervals. We use the Apache Flink event-time metric to ensure equal distancing between adjacent events, and partition data streams into time-bounded arrays.

In fields such as IoT, when collecting sensor data, loss of stream events and non-uniform event transmission is the norm, rather than an exception, especially in regions with lossy networks, or when networks are congested [18]. Missing data, irregular set sizes, or non-uniform streams can often cause instability in systems, or prevent the application of algorithms that rely on the uniformity of data. The data is often forwarded to numerical computation environments such as MATLAB, R or Octave to address data inconsistencies, store them in matrices and perform necessary computations. These methods, however, do not support grouped processing and relational operations, thus requiring coupling with other stream processing systems. SparkR [30], or SciDB-R [24] are examples of systems capable of processing groups in parallel but only offer computation on offline datasets. Most streaming applications require ad-hoc integration of computation environments, which is inefficient and difficult to implement.

To address the need for consistent data for array formation in streaming, we abstract the temporal relation of stream events present in the Flink data streams by processing input data and aligning it to intervals, while filling in missing values through interpolation. This allows us to guarantee users that the arrays provided by our system will always be of a set size, containing elements distributed over constant time intervals, thus eliminating the need to worry about stream form irregularities. Furthermore, creating an all-in-one package for array-based stream processing in Flink eases the implementation difficulty, as knowledge of only one framework is required for implementation, while also not needing to worry about compatibility issues.

Our array framework has potential for use in a wide array of fields in tasks such data pre-processing for machine learning applications. For IoT applications, we provide time-alignment for sensor arrays and allow for performing array queries over groups of uniform sensor data. While we aim to provide an array function interface to facilitate every field where operations on arrays are used, we implement two functions to demonstrate use-cases of our system, as described in the following sections.

2.2.1 Fast Fourier Transform (FFT) Operations

We provide a built-in function allowing users to perform FFT on a stream window which was converted into array form. FFT is commonly used in fields such as Digital Signal Processing (DSP), speech processing and other fields requiring computation in the Fourier domain. Within our framework, users can specify a custom function to be invoked on the transformed array and whether or not the inverse FFT function should be invoked on the output of the user function.

When performing FFT on arrays with overlapping windows, our framework provides aggregators for overlapping values to facilitate commonly used techniques in DSP, such as the overlap-add method [16], where overlapping result values of sliding windows are summed. Users can define custom aggregators for overlapping data in addition to the base ones supported by our system (sum, average, min, max).

2.2.2 Block cyphers

Data reformatting into array structures allows us to use block cyphers to encode parts of streams, distributing potentially unbounded data into finite chunks which can be encrypted and stored. After formatting Flink data streams into arrays of user-specified size, we provide an encryption function using the AES cypher transformation [17] in cypher-block-chaining (CBC) mode. Users can specify a key that is used for encryption and decryption.

While other encryption methods and types are publicly available and in widespread use, we choose to focus on implementing the AES cypher to showcase the encryption functionality of our system. As users have access to the source code, our implementation can easily be adapted to make use of other cyphers and passed as a custom user function to the array function interface.

2.3 Related Work

2.3.1 Stream Processing

Stream processing refers to the real-time processing of data from various sources. With many applications producing high-volume data streams requiring processing, several infrastructures to address that need were developed [2, 7, 10, 11, 22]. SPEs are engines specifically designed to work on streaming data, supporting relational operations on

streams in the style of SQL [26]. Operations such as filtering, mapping and joining of streams are commonly used in conjunction with techniques such as windowing which divide unbounded streams into finite chunks.

SPEs employ a tempo-relational query model where events are associated with a time measure, such as timestamps in Flink [10], or validity intervals in Trill [11]. Conventional SPEs offer support for general, as well as custom queries, processing either individual events or stream slices in the form of stream windows. They, however, provide no measures to ensure data uniformity and time-consistency, making array-based processing very difficult.

Custom user-defined operations can be used to enable array operations, but require complex logic to implement, often requiring workarounds specific to each SPE [23]. Such implementations need to be aware of event timestamps, mapping them to array indices in corresponding windows and filling in all missing values once windows are ready for evaluation. To ensure accurate interpolation of missing values, each window must also be aware of edge events in adjacent windows, requiring maintenance of overlapping values. These techniques require a deep understanding of the SPE framework and require modifications to its back-end operators.

We provide support for array-based operations and move the implementation logic to the Flink back-end, hiding it from users. As such we can fully abstract the temporal data aspect of stream processing, only presenting users with uniform arrays. This allows for the use of temporal queries in Flink, while also enabling array-based operations.

2.3.2 TrillDSP

TrillDSP [23] implements a stream processing model similar to ours focused on DSP tasks like spectral analysis or digital filtering on top of the Trill SPE [11]. They provide an alternative to loosely-coupled systems using numerical computation environments like MATLAB or R, by supporting DSP operations within the Trill package. To achieve that goal, they propose a deep integration of a DSP framework in Trill, operating with a query language providing function abstractions to DSP experts, while integrating with the Trill query language.

The TrillDSP framework provides users with basic signal operations, aggregation and functional signal operations, as well as sampling, upsampling and downsampling methods with interpolation. It also features an extendable “walled-garden” framework

for digital filtering and user-defined window-based DSP operators. As with our system, where streams are converted into arrays before any array-based operations can be performed, TrillDSP first converts streams into signals - special stream types with no overlapping events. To enable DSP tasks in Trill, they sample their signal streams at uniform intervals, filling in missing values through interpolation.

While the underlying methodology of creating uniform data streams used in TrillDSP and our system is similar, our system aims at providing a more general interface allowing DSP tasks in addition to enabling other array-based processing tasks. As mentioned in the previous section, we provide interfaces for spectral analysis methods using FFT and overlap-add techniques in addition to interfaces for tasks in other fields.

Furthermore, Trill and Flink differ in their dataflow implementation in various aspects. Trill associates ingested events with lifetimes, allowing for sampling at intervals based on what elements are active at a given interval. Flink, however, only associates a single timestamp with every event, making direct matching to intervals impossible, as slight fluctuations in event-time would result in most events being misaligned. We thus implement a resampling operator which moves and readjusts the event timestamps to nearby intervals, while interpolating at intervals with no nearby events.

2.3.3 Incremental Sliding Window Computation

Computation of functions over sliding windows can present a large overhead in execution depending on the size of the sliding parameter. If the parameter is set to a low sliding value, large overlaps between windows will be present and each stream event may be evaluated several times. Several techniques for incremental sliding window aggregation have been proposed [29, 27, 28]. One such technique is partitioning windows into smaller “slices”, pre-aggregating the slices and forwarding the aggregated values to the sliding windows each slice is a part of. This eliminates the need to aggregate individual events at each operator, allowing for only slices to be aggregated.

However, such incremental techniques are suited mostly for aggregation of data where data slices produce the same output in each window and the results are not reliant on the remaining window events. Transformations such as FFT cannot forward the same data slices to every window for that reason. Thus, we provide an incremental sliding window computation interface to users where they can forward data to subsequent array window evaluations, containing selected output data of the previous window. This allows for methods such as the Sliding DFT [21] to be implemented.

Chapter 3

Design

In this chapter, we present the requirements of our system and describe its overall architecture and workflow. We give an overview of each system component and detail how data passes through our system. Furthermore, we present the resampling and interpolation algorithm that is used to align data and fill in missing events.

3.1 System Requirements

To enable array-based stream processing operations within Flink, our system needed to meet the following requirements:

Requirement 1 - Uniformly Distributed Stream Events: Flink provides no guarantees of ingested events appearing at regular intervals, with no missing data. To create arrays abstracting the event-time with data sampled at consistent intervals, our system must ensure that events on the input data stream are uniformly distributed at a desired interval rate.

Requirement 2 - Array Function Interface: Providing Flink users with a familiar interface following the style of the Flink query language is crucial to creating a usable framework. Our system needs to provide an interface abstracting the notion of time from events, exposing only arrays of data to users, while seamlessly integrating with the Flink coding model.

Requirement 3 - Performance: With one of the goals of our system being the removal of the overhead in communication when integrating Flink with other computing environments such as MATLAB or R, our system should perform better than implementations making use loose coupling of said environments for array-processing. Fur-

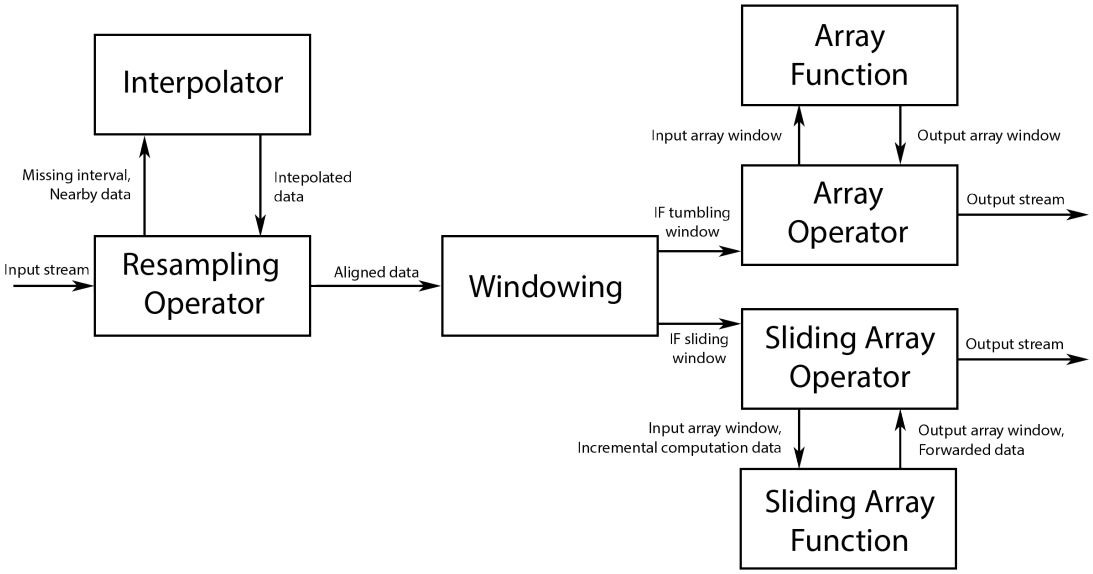

Figure 3.1: Model of the system workflow for array-based processing in Flink.

thermore, the array-based processing model should exhibit performance comparable to standard Flink operations on streaming data.

3.2 System Workflow

To develop a system meeting the aforementioned requirements, the model depicted in Figure 3.1 was adopted. The system is comprised of three main components:

- A sampling module, which through data re-alignment and interpolation ensure that the input data is distributed uniformly,
- Array operator modules for sliding and tumbling windows, which emit arrays used in array functions, as well as handle data forwarding and overlapping data aggregation on sliding windows,
- An extendable function module, which provides an overridable method used to implement user-defined array functions.

As shown in Figure 3.1, the input stream first passes through the resampling module, which is associated with a select interpolator instance. It emits an ordered list of events where timestamps are adjusted to meet the sampling interval of the user while making use of the interpolator to fill in intervals with no data. The uniform and aligned data are then windowed making use of tumbling or sliding windows in Flink.

Depending on the type of window, an appropriate operator is invoked, making use of a user-defined function to operate on the data in each window. The operator transforms the window data into array form, passes it to the array function and emits the result onto the output stream. In the case of sliding windows, the operator also maintains a list of overlapping values emitted by the array function and aggregates them if an aggregator is provided by users. Furthermore, it enables data forwarding where the user function can optionally return an array containing data to be forwarded to the next function call, enabling incremental computation.

The output stream can then be forwarded into a sink, storing the results through a user-specified method. Our system also allows for additional computation on the output stream if further processing is be needed and is compatible with other Flink queries.

3.2.1 Sampling

To meet Requirement 1 described in Section 3.1, the procedure shown in Algorithm 1 was employed. The sampling module uses the algorithm to create uniformly distributed streams to enable array creation. It first moves input events to the closest array slot (interval) based on the time-gap between array elements specified by users *(lines 7-10)*. Intervals are used to index into arrays, where an increment of one time interval corresponds to an array-index increment.

When the interval closest to the event currently being processed changes, the closest event is either emitted onto the output stream or added to an emission buffer *(lines 11-15)*. The emission buffer is used to maintain event order, as it buffers ready-to-emit events until all missing events have been interpolated. The current interval is then adjusted to the interval closest to the current event and every interval that was skipped due to having no nearby events is scheduled for interpolation *(lines 16-19)*.

Lines 20-28 process all intervals waiting to have their events interpolated or emitted, interpolating only if the current event is further than the first event of the interpolation buffer. The interpolation buffer maintains N events closest to the array interval requiring interpolation. Lastly, the interpolation buffer is updated with the most recent event *(line 29)*.

We design the sampling module to work with keyed and non-keyed streams, while also allowing users to specify their own interpolation functions, or choose from a set of pre-defined methods.

Algorithm 1: Resampling and Interpolation algorithm.

```
Input: N ← buffer size
1  begin
       // Buffer of N nearest events for interpolation
2      interpolationBuffer ← empty list of size N;
       // Buffer of elements waiting to be emmited
3      emissionBuffer ← empty list;
       // Align the starting timestamp to the first stream element.
4      currentInterval ← timestamp of first event;
5      closestEvent ← first event of stream;
6      foreach ingested event x do
7          eventInterval ← closest interval to x.timestamp;
8          if eventInterval == currentInterval then
9              if x closer to eventInterval than closestEvent then
10                 closestEvent ← x;
11         else
12             if emissionBuffer is empty then
13                 emitWithWatermark(closestEvent);
14             else
15                 emissionBuffer.add(closestEvent);
16             closestEvent ← x;
17             currentInterval ← eventInterval;
18             foreach skippedInterval do
19                 emissionBuffer.add(new EmptyEvent at skippedInterval);
20         foreach intervalEvent in emissionBuffer do
21             if intervalEvent is EmptyEvent and interpolationBuffer.at(0) is closer
                 than x then
22                 emitWithWatermark(interpolate(intervalEvent));
23                 interpolationBuffer.remove(intervalEvent);
24             else if intervalEvent is not EmptyEvent then
25                 emitWithWatermark(intervalEvent);
26                 interpolationBuffer.remove(intervalEvent);
27             else
28                 break;
           // Update buffer to maintain the last N ingested events
29         interpolationBuffer.update(x);
```

3.2.2 Array Processing

To create arrays, the input stream is first windowed and then passed to user functions in array form. Windowing is done by counting the number of input events, which, due to the uniformly sampled stream, also means that the windows span over a static time-frame. Both keyed and non-keyed streams can be converted into array streams, on which array functions can be invoked.

To adhere to the standard Flink query language and meet Requirement 2, array streams provide an "apply" method, which passes user-specified array functions to an operator. The array function interface abstracts the notion of time and underlying array creation logic and allows users to focus on creating functions that use arrays as input.

We provide support for tumbling and sliding windows in the form of separate operators, providing a function interface that exposes the input stream in array form.

3.2.3 Sliding Windows Operations

To enable efficient and usable array operations on sliding windows, we employ a Sliding Window Operator that allows for incremental computation, as well as aggregation of overlapping output arrays.

Incremental computation is achieved by providing users with an interface where their array function output can return partial results done on data overlapping with future windows in addition to the full result array. This data is then forwarded to the next user function call, where it can be used to avoid computation on already processed array sections.

Furthermore, sliding windows result in overlapping events if the slide parameter is lower than the window size. This results in duplicate elements being emitted to the output stream if no aggregation is done within user functions. To address that issue we provide a function interface allowing for an aggregator to be specified which aggregates overlapping events in output arrays. This allows for techniques such as overlap-add, or overlap-save to be applied.

Chapter 4

Implementation

In this chapter, we detail how all system components were implemented and provide code samples showing example uses of all operators. We discuss the differences in implementation for grouped and non-grouped stream processing. We then provide an overview of how operations on sliding windows are performed, providing examples of overlapping data aggregation and describe the framework for incremental computation. In the subsequent section, we describe the implementation of the FFT and block cypher use cases and highlight our array function interface. Lastly, we give an overview of our system's limitations and challenges faced during implementation.

4.1 Data Alignment and Interpolation

To facilitate the creation of arrays with uniformly-sampled data, we create a `resampling operator` that selects data closest to each sampling interval and uses a user-specified `interpolator` to fill in missing data.

4.1.1 Resampling Operator

The resampling operator processes a Flink `DataStream` and outputs a new uniform `DataStream` with events at regular intervals without missing data. When an event is ingested by Flink, it passes through a `resampler`, which extracts its event-time timestamp and aligns it to the nearest interval point. When multiple events would be assigned to the same time interval, only the nearest one is emitted onto the resulting data stream. Figure 4.1 illustrates the data alignment procedure. The resampling operator can be instantiated by calling `resample` on a data stream with the sampling

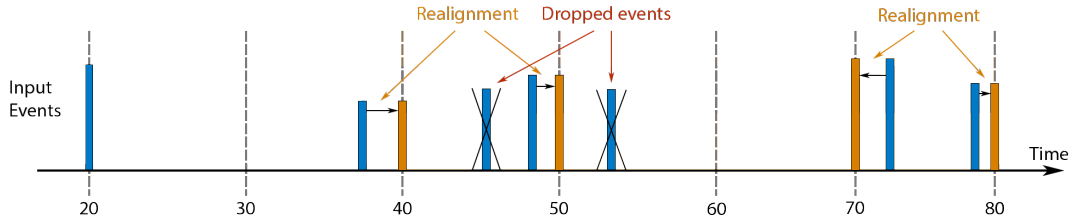

Figure 4.1: Resampling of misaligned input data and dropping events that are not closest to an interval. Blue bars represent input events, while orange ones show how they are realigned. Redundant events are crossed out and labeled as “dropped”.

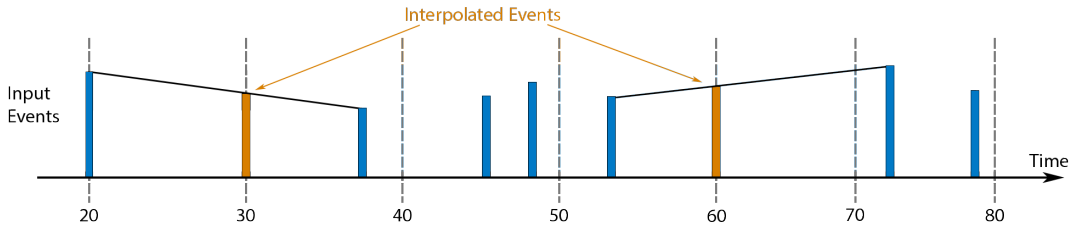

Figure 4.2: Linear interpolation at intervals with missing data. Blue bars represent input events, while orange ones show interpolated events at intervals with missing data. The two nearest input events are used for linear interpolation.

interval in milliseconds and an `interpolator` as parameters. An optional parameter `samplingWindow` in milliseconds can be specified, which drops any events with an event-time further than `samplingWindow` apart from the closest sampling interval. Timestamps and watermarks of output events are set to match the sampling intervals.

4.1.2 Interpolation

As the formation of arrays requires all intervals to contain data, an `interpolator` is used to fill in any missing events in the data stream. Whenever the resampler detects a gap within the data where no events were assigned to an interval, the `interpolate` function is used to create an interpolated event. Our implementation provides three native interpolators, as well as an interface where users can define their own.

Each interpolator maintains a buffer containing a user-specified number of timewise nearest events. When an event further than the first event in the buffer is ingested, the `interpolateAndCollect` function is called, which creates and emits a new event onto the resulting data stream. To maintain event order, all ingested events are buffered

until the interpolation is done and emitted only after its completion. All events are emitted to the output stream with a timestamp and watermark corresponding to the interval at which they have been interpolated to facilitate the usage of time-reliant operators later in the dataflow graph of a Flink program.

In our system we provide the following in-built interpolators:

- Linear interpolator - uses two values nearest to the sampling interval to determine the interpolated value through a linear function that passes through the two nearest events (shown in Figure 4.2).
- Averaging interpolator - uses a user-specified number of nearest elements and averages them to fill in the missing data point.
- Default interpolator - fills in every missing data point with a default value.

The following code sample shows an application of the resampling operator in conjunction with an averaging interpolator. We specify the sampling interval as 10ms, a sampling window of 3ms and an interpolation buffer size of 4 events.

```
1 DataStream<Double> source = ...
2
3 AveragingInterpolator<Double> interpolator = new AveragingInterpolator<>(4);
4 DataStream<Double> output = source
5         .resample(10L, interpolator, 3L);
```

4.1.3 Keyed Resampling

When dealing with streams of multiple sources, or in cases where different data types are being captured and sent as separate events, the stream can be partitioned into groups based on a specified key-field present in each event. To facilitate the use of the resampling operator and interpolators, our system provides keyed versions, supporting usage on keyed streams.

The keyed resampling operator maintains a Flink `value state` for every key, containing relevant buffers and variables. The data present in the state is passed to the resampler and interpolator on every call. The `value state` maintains a `DataStorage` object which holds the state variables needed for resampling and interpolation. Flink's `keyed state` interface handles state management per key, removing the need to manually buffer all resampling data and match input events to relevant keys. Furthermore,

making use of the Flink native support for keyed state allows the resampling to be parallelised by the job manager with several keyed resampling operator instances being invoked, each handling input data for a different subset of keys.

The sample code below shows the usage of the keyed resampling operator. It takes an additional parameter `dataField` that tells the operator which data field to use for interpolation to differentiate data values from keys.

```
DataStream<Tuple2<Double, Long>> source = ...

KeyedAveragingInterpolator<Tuple2<Double, Long>> interpolator = new
    KeyedAveragingInterpolator<>(4);
KeyedStream<Tuple2<Double, Long>, Long> out = source
       .keyBy(t -> t.f1)
       .keyedResample(10L, interpolator, 0);
```

4.2 Array Operator

We provide an operator allowing users to implement functions taking `ArrayLists` as parameters, which produce transformed `ArrayList` outputs. The operator turns Flink stream windows into `ArrayLists` and passes them to the user function whenever a window is full. The code below shows a standard use pattern of the array operator, making use of the resampler.

```
DataStream<Double> source = ...

AveragingInterpolator<Double> interpolator = new AveragingInterpolator<>(4);
DataStream<Double> output = source
       .resample(10L, interpolator)
       .countWindowAll(256L)
       .toArrayStream()
       .applyToArray(new UserArrayFunction());
```

After resampling, the stream is windowed by using a counting window of size 256 events. As the events are sampled at a 10ms interval, the windows will span events across 2560 milliseconds. To gain access to the array operator, `toArrayStream` is called. `applyToArray` takes as parameters a user-defined class which extends the `ArrayWindowFunction` abstract class and overrides its `userFunction` method. The same use pattern can be applied to keyed streams.

The operator uses a `ListState` to collect windowed values and transform them into arrays. As `ListState` is a keyed state, when instantiating the operator on a non-keyed stream, the `countWindowAll` function transforms the stream into a native Flink `AllWindowedStream` type object, mapping every input event to the same key. This allows both Keyed and Non-Keyed streams to make use of the Flink `Keyed State` for array formation.

4.3 Sliding Windows

As described in Section 2.1.3.1, Flink supports sliding windows, which are evaluated every time a period specified by the slide parameter passes. Depending on the configuration, an event can be assigned to multiple windows, causing the same events to be processed several times. To enable incremental computation on sliding windows, sliding array functions provide an additional input parameter containing partial results of the previous window evaluation. To make use of the incremental computation interface, user functions on sliding windows must return a tuple containing both the full, as well as the partial array output which is then passed to the next user function call and can be used for incremental computation.

4.3.1 Overlapping Data Aggregation

If the user function does not aggregate its output when sliding windows overlap the resulting data stream may contain multiple processed instances of an event. To enable methods such as overlap-add, the overlapping output events can be aggregated by using an `SlidingAggregator`. The aggregator is passed to the `SlidingArrayOperator` by calling `applyToSlidingArray` with an extra parameter containing an aggregator class that extends the `SlidingAggregator` abstract class.

In our initial implementation of overlapping data aggregation, when profiling the operator's performance, a large overhead was identified due to the underlying ArrayList structure. Overlapping values which were already aggregated and emitted were being removed from the aggregation buffer, due to the `ArrayList.remove()` operation having a time complexity of *O(n)*. Thus, the aggregation operation increased the sliding window operator processing time by a significant margin. To address the issue, we instead opted to implement an aggregation algorithm implementing a variation of a circular array by overwriting previous data instead of evicting it from the list. Figure

4.3 shows how overlapping values are aggregated with a sum aggregator.

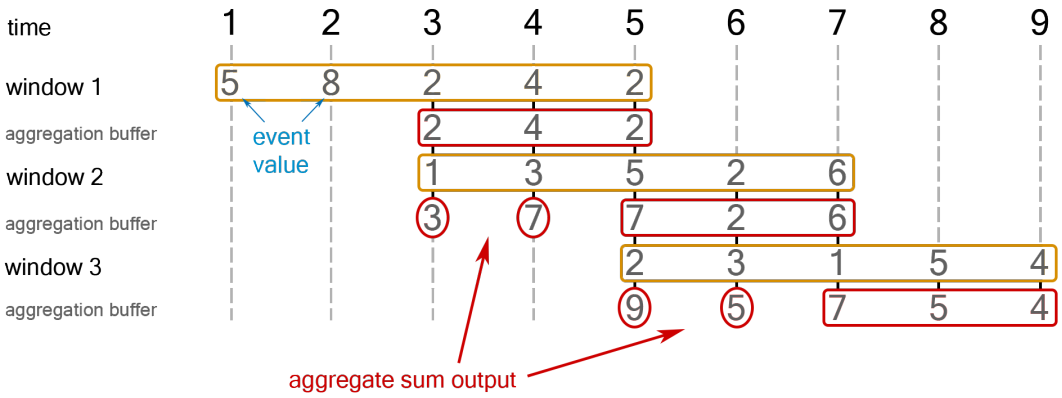

Figure 4.3: Example of overlapping value aggregation of three sliding windows. The aggregation buffer in the red rectangle shows what values are stored for overlap aggregation after window evaluation.

4.4 Array Functions

Our framework provides two built-in methods for array processing in addition to an extendable abstract array function allowing for user-defined functions.

4.4.1 Fast Fourier Transform

The computation of the frequency domain of a signal through FFT is a common use case in DSP tasks. Our FFT function allows for a common work pattern in DSP to be applied, where a signal is converted into its frequency domain, a function modifies the computed FFT results and the result is passed through an inverse FFT function. For the function implementation, the Apache Commons [5] FFT function using the Cooley-Tukey [15] algorithm was used.

To define the custom function applied in the frequency domain, users need to override the `userFunction` method within the FFT class with their implementation. The default `userFunction` is an identity function.

```
// Input stream of Double type values and String type keys
DataStream<Double,String> source = ...

LinearInterpolator<Double> interpolator = new LinearInterpolator<>();
DataStream<Complex> output = source
        .keyBy(t -> t.f1)
        .keyedResample(25L, interpolator, 0)
        .countWindow(1028L)
        .toArrayStream()
        .applyToArray(new DoFFT());
```

When computing FFT on sliding windows, if a method requiring aggregation of overlapping events is needed, the sliding window array interface can be used. For the overlap-add method, users must simply specify an adder which aggregates all overlapping values.

```
DataStream<Double> source = ...

LinearInterpolator<Double> interpolator = new LinearInterpolator<>();
DataStream<Complex> output = source
        .countWindowAllSlide(1028L, 128L)
        .toArrayStreamSliding()
        .applyToArraySliding(new DoFFT(), new Adder());

...

public static class Adder implements SlidingAggregator<Complex>{
   @Override
   public Complex aggregate(Complex val1, Complex val2){
      return val1.add(val2);
   }
}
```

4.4.2 Block cyphers

Block cypher encryption is implemented through the use of the Java cypher class. The stream arrays are passed to the function, producing an encrypted output array. The cypher used for encryption is AES in CBC mode. By manipulating the event-time at the stream source, the resampling operator can be used to add padding to output data

to create encrypted blocks of uniform size, while maintaining segregation of stream data into separate blocks. By emitting a part of an array block with timestamps in incrementing intervals, and then skipping the remaining timestamps that fall into that array, the `DefaultInterpolator` can be used to pad the missing array segments. The following code segment shows an example usage of our block cypher function.

```
DataStream<Byte> source = ...

DefaultInterpolator<Byte> interpolator = new DefaultInterpolator<>(0);
DataStream<Byte> output = source
        .resample(1L, interpolator)
        .countWindowAll(1000L)
        .toArrayStream()
        .applyToArray(new EncryptArray());
```

4.4.3 User-Defined Functions

In addition to providing support for the above use-cases, the array function interface can be used to implement any custom array function on both tumbling and sliding array windows. The already shown `applyToArray` and `applyToArraySliding` methods of `ArrayStreams` can be used to invoke custom array functions.

We provide abstract classes `ArrayWindowFunction` and `SlidingArrayWindow Function` which must be extended by user classes that override the `userFunction` method. The `userFunction` is given the array data of each window as input in the form of an `ArrayList`, expecting an `ArrayList` output. In the case of sliding windows, the user function can return a tuple of both the full array output, as well as the array part to be passed to the next function call for incremental computation.

4.5 Challenges

During implementation and integration of our framework with Flink, we faced several challenges.

4.5.1 Generic Types and Type Erasure

The Flink back-end operates entirely using Java generic types, only inferring input and output types based on user-defined functions and input stream sources. As such,

extraction of data types when performing arithmetic for interpolation, or providing default values was often very difficult. Type reference objects and type information needed to be passed along the entire operator call chain to ensure safe casting and prevent errors caused by operations on incorrect data types.

Furthermore, additional care had to be taken to ensure that type information was preserved after every operator. Type erasure caused significant problems when attempting to merge our operators into the existing Flink engine implementation. Thus, testing was done to ensure that all explicit type casts used to preserve type information were valid.

4.5.2 Operator Output Retrieval

As the functional interfaces present in Flink do not allow for retention of results of stream functions in the operator invoking the function, the implementation of result forwarding and aggregation of overlapping events in sliding windows presented a challenge. To circumvent the restrictions imposed by Flink, we implemented a new function structure allowing us to retrieve results of functions applied to sliding window arrays. We created new wrappers and interfaces specifically aimed at sliding window functions which pass the output data back to the operator to forward and aggregate overlapping data before emitting it onto the resulting stream.

4.6 Limitations

With some of our implementations not necessarily following the Flink suggested workflow, some limitations of our system arose during integration with other Flink components.

4.6.1 Sliding Window Overlap Computation

In our current implementation, overlapping data computation is done by computing overlapping array indices of arrays emitted by user functions. This, however, imposes a limitation on the system, where output arrays must be of the same length as input arrays. This presents no issues when implementing DSP processing tasks with FFT, making use of the overlap-add method. However, were a user function to output larger or smaller arrays than the input ones, the overlapping data computation would fail

and users would need to resort to alternative aggregation techniques in their function. Doing so, however, is often very difficult within the original Flink framework.

4.6.2 Type Conversion to ArrayList and ListState Updating

Apache Flink maintains lists of window contents in the Java `Iterable` structure. As such, conversion to either an `ArrayList` or `Array` structure requires *O(n)* computation time. This causes an overhead in computation when creating arrays, which is most apparent on sliding window computation with small sliding parameter sizes, as several arrays need to be created consisting of many overlapping events.

Furthermore, when computing on sliding windows created through global windows, we have to maintain a list of elements through the Flink `ListState` and remove elements which are not part of the next sliding window. This presents an issue as the `ListState` in Flink does not support a `remove` operation. Therefore, we must clear the state and re-add every element that must remain in the window on every evaluation. This issue is also present in the native Flink window operators when `evictors`, are used to remove elements from a window according to user-specified parameters. The native Flink `count window` implementation uses `evictors` to maintain the global window state, incurring a similar, or greater performance overhead than our implementation.

4.6.3 Parallelism

As all input events must first pass through the same resampling operator instance to create a single, uniformly sampled stream, the resampling operation is inherently non-parallel. This limits the performance of the system, as it prevents operations from being scaled through distributed computation.

However, parallel execution can be used to improve performance when computing on keyed streams. As resampling is done separately for every stream key, the operator and all operators applied subsequently can be parallelised.

Chapter 5

Evaluation

In this chapter, we describe how we evaluated our system on whether it meets Requirement 3 - performance described in Section 3.1. We compared it against loosely-coupled array-based processing within Flink, integrating calls to the MATLAB computation environment for array processing, as well as against native Flink operators to estimate the overhead of our operators.

We focus on evaluating each operator separately, estimating the computation costs associated with them. The evaluation is done for keyed and non-keyed streams, using both tumbling and sliding window allocators.

Experimental Setup: The experiments were run on an Ubuntu 20.04 instance consisting of an Intel i7-8700K CPU @ 4.80GHz and 32GB of RAM @ 3.2GHz

5.1 Loosely-Coupled Systems in Flink

In our work, we state that integration of an outside numerical computation environment such as MATLAB in Flink is not a usable solution due to the communication overhead presented when invoking MATLAB functions. To confirm that hypothesis we used the MATLAB Java engine to construct an experiment in which we called MATLAB functions from within a Flink operator.

As our primary interest lied in determining the communication cost, we constructed a uniformly sampled stream of 15 million input events, requiring no pre-processing before array construction. The randomly generated `double` type events were windowed into tumbling windows and passed to MATLAB where they were processed by the FFT and inverse FFT functions. The resulting data was emitted back to the stream without any aggregation.

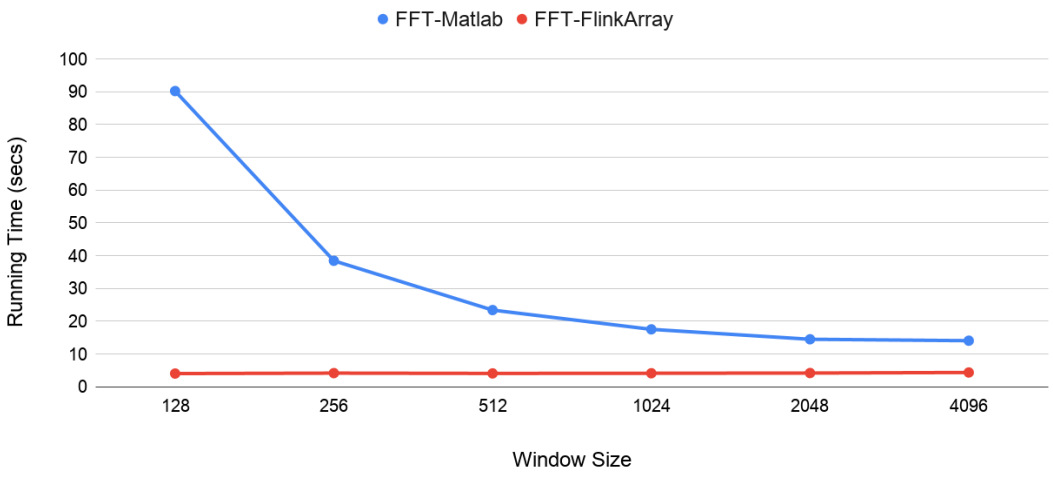

Figure 5.1: FFT over tumbling count window comparison between Flink array-processing and integration with MATLAB.

The results were compared to the same FFT transformations using our system, first performing resampling to verify the uniformity of the input stream, creating an array and using our FFT array function to compute the Fourier transform, as well as its inverse. The timestamps for the input stream were generated incremented by 1ms for every event on the stream. We express our query in Flink as follows:

```
1 LinearInterpolator<Double> interpolator = new LinearInterpolator<>();
2 DataStream<Double> output = source
3         .resample(1L, interpolator)
4         .countWindowAll(windowSize)
5         .toArrayStream()
6         .applyToArray(new DoFFT());
```

Figure 5.1 shows the execution time on differently sized windows. As shown, our system performs significantly better than the loosely-coupled integration of MATLAB in Flink. The communication overhead is reduced when performing computation over larger window sizes, as data is passed to MATLAB less often. However, our system still performs better even at larger window sizes. We thus meet the requirement of our system outperforming loosely-coupled systems and confirm our hypothesis that such systems incur a large communication overhead, rendering them ineffective.

5.2 Performance Comparison to Native Flink Operations

To measure the overhead of our system on the Flink engine we compare its performance to native Flink aggregate operations, focusing on individual operators. Our system runs a resampling operator before other query operators are invoked, as well as requires data processing to expose arrays to users. The additional steps in execution add a processing-time overhead to Flink which we aim to minimize. In this section we evaluate the costs of our system and how it performs in comparison to native operators in Flink, focusing on aggregation.

5.2.1 Resampling Operator Performance

To estimate the cost of aligning input events and interpolating missing values, we run the resampling operator on three different input stream varieties for different amounts of inputs. The following stream types were used as experiment inputs:

- uniformly sampled stream, no missing values
- non-uniformly sampled stream, no missing values, requires only data alignment
- uniformly sampled stream, 1/3 missing values requiring interpolation

A resampler was invoked on every stream, using a linear interpolator. We express our query as follows:

```
1 LinearInterpolator<Integer> interpolator = new LinearInterpolator<>();
2 DataStream<Integer> output = source
3       .resample(10L, interpolator);
```

As shown in Figure 5.2, the cost of resampling does not change with stream characteristics and the cost increases linearly with the number of events processed. This shows that the majority of the performance cost of resampling comes not from alignment and interpolation, but from reading all input data and determining whether interpolation is needed.

Moreover, as shown in Figure 5.3, we observe an approximately 30% increase in runtime when invoking a windowed aggregation query with the resampling operator, compared to invoking the same query without it. Subtracting the runtime of both queries and comparing it to the input event counts shows that the resampling operator cost in the aggregation query was approximately 40-60 ms per million input events,

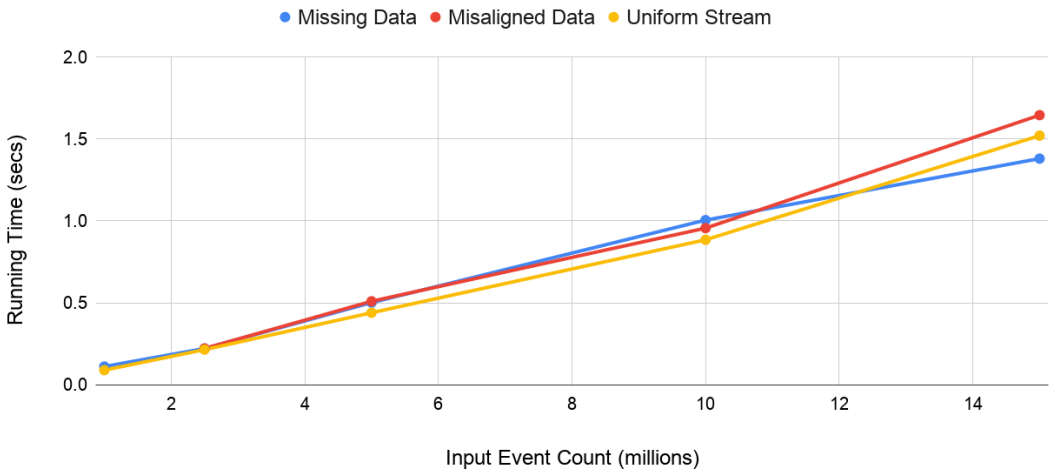

Figure 5.2: Resampling operator runtime comparison on an uniform stream, stream with misaligned data and a stream with missing data.

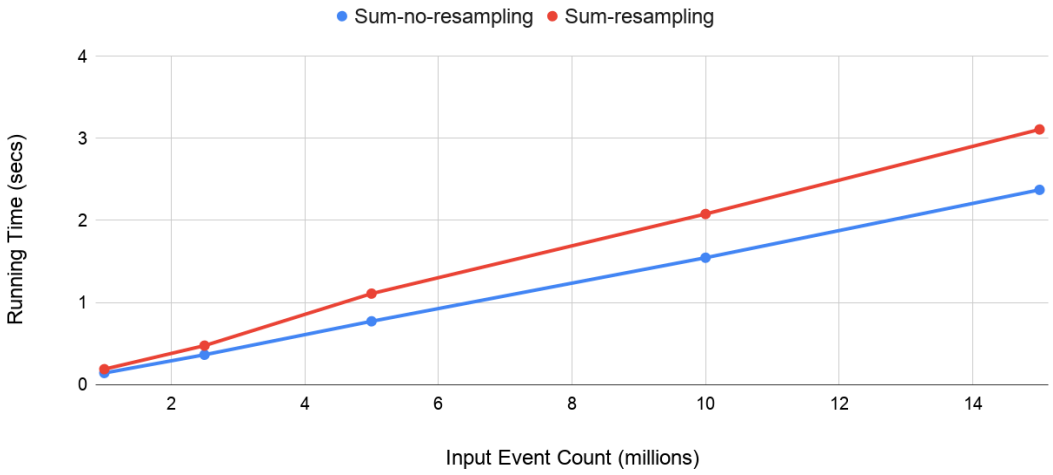

Figure 5.3: Sum aggregation query performance with and without event resampling.

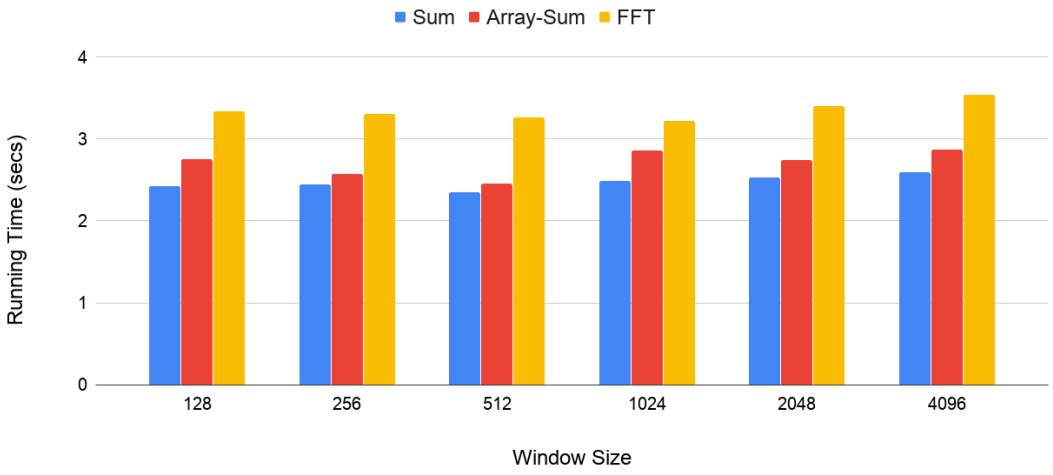

Figure 5.4: Performance comparison between array operators FFT, array sum, as well as a Flink default sum aggregator and a on non-keyed streams of tumbling count windows.

which is slightly lower than the results in Figure 5.2 suggest. This is likely due to computation optimization in Flink when performing computations and working with multiple operators. These results show that the resampling operator offers competitive performance and does not present a significant bottleneck in the Flink dataflow.

5.2.2 Tumbling Window Evaluation

We evaluate the performance of our system on tumbling windows with keyed and non-keyed streams. We again make use of the aggregate sum operator as a native Flink function benchmark and compare it against an array function sum, which computes the same aggregate, but instead using arrays as inputs. We also compare the performance to FFT array functions to observe the difference in performance when performing more complex calculations on array streams. Input streams of 15 million uniformly sampled events were used for evaluation.

5.2.2.1 Non-Keyed Streams

As stated, we compared two array functions, as well as a native Flink aggregate function. Figure 5.4 presents the running time of each function on windows of different sizes. We note that window size has no impact on performance, with all functions achieving similar running times independent of window size.

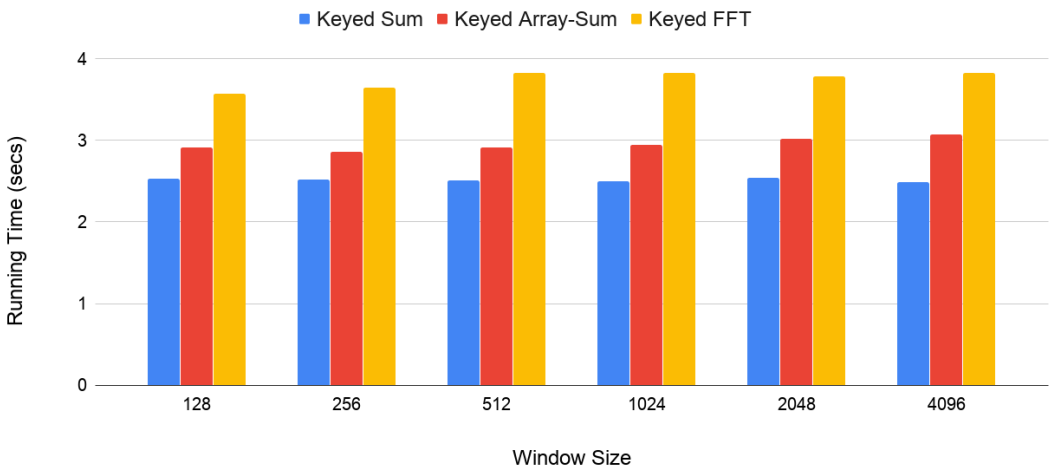

Figure 5.5: Performance comparison between array operators FFT and array sum, as well as a Flink default sum aggregator on keyed streams of tumbling count windows.

The average increase in computation time of the array-sum function was roughly 10%, while the FFT operator was 20% slower. The difference in the sum operator mostly stems from the overhead in type conversion from the Java `Iterable` construct used by Flink into the `ArrayList` format passed to the user function. FFT runtime is higher due to the increased function complexity but still achieves competitive runtimes.

5.2.2.2 Keyed-Streams

The experiments on keyed streams were executed on input streams partitioned into 100 groups where input events were evenly split amongst all groups. Similar to our non-keyed stream evaluation, the same three function operators were executed on the input streams. Results are shown in Figure 5.5. As with non-keyed results, the window size did not impact the runtimes. We do however observe an 10% average increase on array-based operations when compared to their non-keyed variants, while the native Flink sum retained the same performance. This overhead is likely caused due to the need to maintain internal list data structures that scale with the number of groups.

As noted in Section 4.6.3, keyed array stream operations can be parallelised with each operator instance handling a subset of keys/groups. We evaluate our system for different parallelism settings in Flink, which dictate how many instances of each operator are created. Figure 5.6 shows the runtime results of invoking the FFT array function with different parallelism settings with window size 1024. Interestingly we observe that once set to create more than 2 instances of each operator, the performance

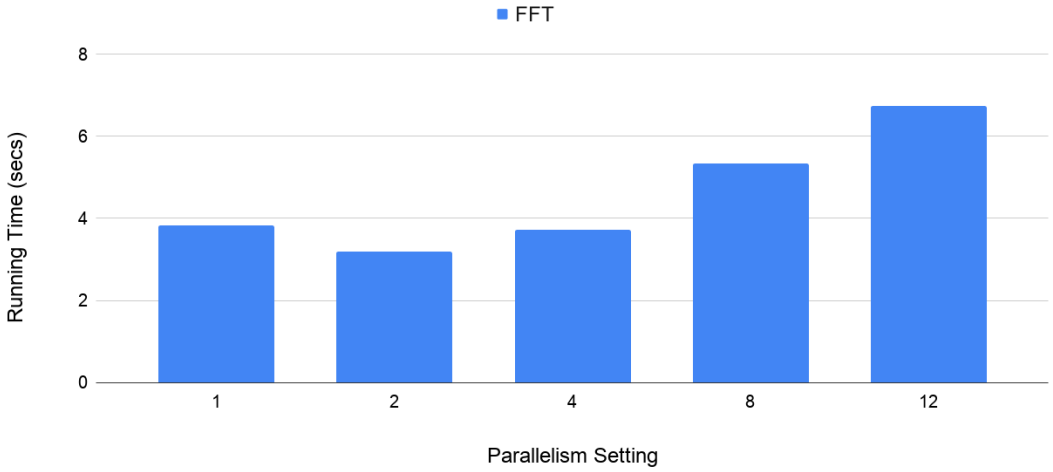

Figure 5.6: Graph showing running time of the array FFT operator with different degrees of parallelism set through the Flink function `env.setParallelism(n)`, with window size 1024.

starts degrading. This could be a result of component limitations of our experimental setup, or the overhead of managing internal data structures exceeds the gain of parallelising execution. Furthermore, all events were ingested into the system from a single input stream and then partitioned into groups. If multiple stream sources were used, higher parallelism would likely yield better results.

5.2.3 Sliding Window Evaluation

To evaluate the performance of our system on sliding windows, we implemented an incremental sum operator which used data forwarding to forward already computed parts of the aggregate to subsequent windows. Figure 5.7 shows the results of running functions on a stream of 15 million events, window size 256 and various slide sizes.

We notice that the incremental computation of aggregates had little to no effect on the runtime. Upon further inspection of the results using a CPU profiler, we find that the aggregate computation only comprises a very small part of the running time, with most of the time being the result of array formation and window list state management. The runtime increases as we decrease the slide size, causing arrays to be formed more often. However, our aggregate implementation still outperforms the Flink native sum using the standard window operator on sliding count windows. This is mainly due to the limitation mentioned in Section 4.6.2 as an `evictor` is used to manage the window state, incurring large overheads. Our system achieves better performance by integrat-

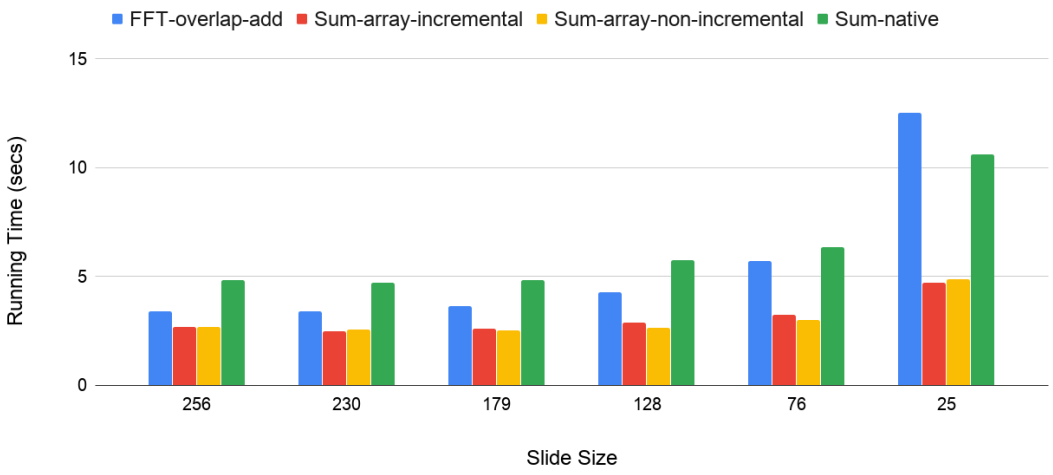

Figure 5.7: Performance comparison between the array functions FFT with overlap-add, Sum and Incremental Sum against the native Flink Sum operator on sliding count windows.

ing state updating into the sliding window array operator without using an `evictor`.

While array creation presents a major part of the runtime of a simple array aggregate, when observing the performance of a higher complexity procedure such as FFT overlap-add, the execution of said procedure starts to be the bottleneck in execution as the full array needs to be re-evaluated on every slide. Thus, the execution of more complex functions could benefit significantly from the usage of our incremental computation framework. In the case of FFT techniques such as the Sliding DFT [21] can be used to increase performance.

5.3 Summary of Results

We evaluated our system on its performance in comparison to a loosely-coupled integration of Flink and MATLAB, as well as conducted per-operator experiments to determine the performance cost of our operators compared to ones supported natively by Flink.

Loosely-coupled Systems: As shown in Figure 5.1, our system outperformed the coupling of Flink and MATLAB significantly at lower window sizes where more communication was needed. At higher window sizes, our system still performed better, but the difference in performance narrowed. We conclude that our system meets the initial

performance goals aiming to perform better loosely-coupled systems in Flink.

Resampling Operator: We show that the cost of resampling does not change given different types of input streams, where none or many events need to be interpolated, or realigned (Figure 5.2). We estimate the runtime cost of resampling at around 40-60ms per 1 million input events, yielding an approximately 30% performance overhead on a simple Flink aggregate function, as shown in Figure 5.3.

Tumbling Windows: Our array-based processing operator incurs a 10% cost in runtime when performing an array aggregate over tumbling windows as compared to the native Flink sum (Figure 5.4). This overhead is attributed to back-end data type conversions. When performing keyed computations, the runtime is increased by an additional 10% (Figure 5.5), but can be improved by increasing the parallelism of Flink operators. Due to our experimental setup, as shown in Figure 5.6, the performance improved only at a parallelism setting of 2 and started degrading afterwards. This result would likely change in a more highly distributed processing setup.

Sliding Windows: We show that our sliding window operator improves on the native Flink aggregation operator on sliding count windows by modifying the window back-end implementation. However, as shown in Figure 5.7 by the array sums, performance still suffers due to type conversion to arrays at lower slide sizes. We do not see a difference in runtime compared to incremental and non-incremental aggregation methods, as the runtime is dominated by type conversions. A more complex algorithm, such as Sliding DFT [21], would need to be used to show the benefits of incremental computation.

Chapter 6

Conclusion

This project was undertaken to bridge the gap between relational stream processing in stream processing engines (SPEs) and array-based operations performed in numerical computation environments such as MATLAB and R. Loose coupling of SPEs and MATLAB/R is often used to gain access to functionalities of both systems, but that incurs large processing costs due to the communication overhead when passing data from one environment to the other. Thus, we present a solution in the form of an extension to the Apache Flink [10] SPE, enabling array-based operations such as the Fast Fourier Transform (FFT) to be performed on streams, eliminating the communication overhead. We build on top of the Flink framework, retaining the ability to execute all relational queries supported by the native Flink implementation while providing our extensions for array-based processing.

To enable array-based processing in Flink, we implemented several operator modules responsible for creating uniform streams through data resampling and missing value interpolation. The implementation provides users with an interface allowing for custom array function creation, adhering to the Flink coding style by using an `apply` function, through which user functions can be invoked. As proof of concept, we provide two built-in array functions: FFT for use in signal processing and a block cypher implementation for encryption. Furthermore, we provide interfaces enabling incremental computation on sliding windows, as well as aggregation of overlapping data.

Our performance analysis showed that our system achieves significantly better performance than loosely-coupled systems which make use of the MATLAB Java engine. The experiments confirmed that data transfer to MATLAB presents a significant performance bottleneck. Furthermore, our results showed that our system manages to achieve competitive running times in comparison to native Flink operations. Our tests

for incremental computation performance did not produce definitive results, likely due to the low complexity of our chosen incremental function. The implementation of a more advanced incremental operation on sliding windows such as the Sliding DFT [21] in future work is required to show the impact of incremental computation on sliding windows in our system.

While our system achieved the goals of creating a functional framework with competitive performance in Flink, we faced many challenges and limitations during implementation. The main obstacle was working around the Flink dataflow and underlying data structures. Flink operates on a forward-only dataflow model, where an operator processes a single window and adds results to either the output stream or a separate managed state. When the operator moves onto the next window, it has no awareness of past windows inputs or results. This made the implementation of data forwarding, overlapping aggregate computations, as well as resampling difficult, as all those operations required knowledge of timestamps and contents of already processed events/windows. To address this framework compatibility issue, significant modifications and workarounds in the Flink engine, as well as the implementation of several custom data buffers was required, which were detrimental to the system's performance. Furthermore, as we worked with the Flink list state, we needed to convert all states from the `Iterable` object type used by Flink to `ArrayLists` which are passed to user functions. This incurred significant type conversion overheads that were visible in our sliding window evaluation.

Despite the limitations, we believe that our system provides an important feature within the Flink SPE while maintaining competitive performance. Array-based processing on streams is a topic relevant in many areas such as IoT, machine learning or digital signal processing. We have shown that integration of array-based processing at high performance is possible in Flink. With further modifications to the Flink backend, addressing the aforementioned limitations, our system could be improved even further.

6.1 Future Work

To address the limitations of our system and further verify the performance impact of our incremental computation system, the following work is to be undertaken in the future:

- Implement a custom Flink state, making use of either `Arrays` or `ArrayLists` as the underlying structure to eliminate the need for type conversion in our operators.
- Provide an additional built-in array function for Sliding DFT [21] to showcase and evaluate the performance of our incremental computation features.
- Parallelise the execution of resampling on non-keyed streams by maintaining a synchronized buffer which maintains data order and the output stream uniformity property.
- Refactor, restructure as an importable package and further test our system on larger stream processing implementations to prepare for making our project open-source.

Bibliography

[1] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. The design of the borealis stream processing engine. In *Cidr*, volume 5, pages 277–289, 2005.

[2] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. The design of the borealis stream processing engine. In *Cidr*, volume 5, pages 277–289, 2005.

[3] Tyler Akidau, Alex Balikov, Kaya Bekiroglu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: Fault-tolerant stream processing at internet scale. In *Very Large Data Bases*, pages 734–746, 2013.

[4] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. 2015.

[5] Apache. Apache commons math 3.6.1 api, 2016.

[6] Apache. Apache flink documentation, v1.11, 2020.

[7] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. Stream: The stanford data stream management system. In *Data Stream Management*, pages 317–336. Springer, 2016.

[8] Paul Brown. Overview of scidb: large scale array storage, processing and analysis. pages 963–968, 01 2010.

[9] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. State management in apache flink®: Consistent stateful distributed stream processing. *Proc. VLDB Endow.*, 10(12):1718–1729, August 2017.

[10] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[11] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John C. Platt, James F. Terwilliger, and John Wernsing. Trill: A high-performance incremental query processor for diverse analytics. *Proc. VLDB Endow.*, 8(4):401–412, December 2014.

[12] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Cetintemel, Ying Xing, and Stanley B Zdonik. Scalable distributed stream processing. In *CIDR*, volume 3, pages 257–268, 2003.

[13] 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)*, pages 1789–1792, 2016.

[14] M. Chock, A. F. Cardenas, and A. Klinger. Database structure and manipulation capabilities of a picture database management system (picdms). *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-6(4):484–492, 1984.

[15] James W. Cooley and John W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Math. Comput.*, 19:297–301, 1965.

[16] R. Crochiere. A weighted overlap-add method of short-time fourier analysis/synthesis. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 28(1):99–102, 1980.

[17] Joa Daor, Joan Daemen, and Vincent Rijmen. Aes proposal: rijndael. 10 1999.

[18] Guillaume Ferré. Collision and packet loss analysis in a lorawan network. In *2017 25th European Signal Processing Conference (EUSIPCO)*, pages 2586–2590. IEEE, 2017.

[19] Lewis Girod, Kyle Jamieson, Yuan Mei, Ryan Newton, Stanislav Rost, Arvind Thiagarajan, Hari Balakrishnan, and Samuel Madden. Wavescope: a signal-oriented data stream management system. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 421–422, 2006.

[20] S Idreos, F Groffen, N Nes, S Manegold, S Mullender, and M Kersten. Monetdb: Two decades of research in column-oriented database. *IEEE Data Engineering Bulletin*, 2012.

[21] E. Jacobsen and R. Lyons. The sliding dft. *IEEE Signal Processing Magazine*, 20(2):74–80, 2003.

[22] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 239–250, New York, NY, USA, 2015. Association for Computing Machinery.

[23] Milos Nikolic, Badrish Chandramouli, and Jonathan Goldstein. Enabling signal processing over data streams. In *SIGMOD 2017, May 14-19, 2017, Chicago, Illinois, USA*. ACM, May 2017.

[24] M. Stonebraker, P. Brown, D. Zhang, and J. Becla. Scidb: A database management system for applications with complex analytics. *Computing in Science Engineering*, 15(3):54–62, 2013.

[25] Michael Stonebraker, Jacek Becla, David DeWitt, Kian-Tat Lim, David Maier, Oliver Ratzesberger, and Stan Zdonik. Requirements for science data bases and scidb. 10 2009.

[26] Michael Stonebraker, Uundefinedur Çetintemel, and Stan Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Rec.*, 34(4):42–47, December 2005.

[27] Kanat Tangwongsan, Martin Hirzel, Scott Schneider, and Kun-Lung Wu. General incremental sliding-window aggregation. *Proceedings of the VLDB Endowment*, 8:702–713, 02 2015.

[28] Jonas Traub, Philipp Grulich, Alejandro Cuéllar, Sebastian Breß, Asterios Katsifodimos, Tilmann Rabl, and Volker Markl. Scotty: Efficient window aggregation for out-of-order stream processing. 04 2018.

[29] Jonas Traub, Philipp M Grulich, Alejandro Rodríguez Cuéllar, Sebastian Breß, Asterios Katsifodimos, Tilmann Rabl, and Volker Markl. Efficient window aggregation with general stream slicing. In *EDBT*, pages 97–108, 2019.

[30] Shivaram Venkataraman, Zongheng Yang, Davies Liu, Eric Liang, Hossein Falaki, Xiangrui Meng, Reynold Xin, Ali Ghodsi, Michael Franklin, Ion Stoica, et al. Sparkr: Scaling r programs with spark. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1099–1104, 2016.

[31] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.