Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Bachelor's Thesis Nr. 197b

Systems Group, Department of Computer Science, ETH Zurich

Implementation of a Benchmark Suite for Strymon

by

Nicolas Hafner

Supervised by

Dr. John Liagouris
Prof. Timothy Roscoe

November 2017 - May 2018

Informatik
Computer Science

Abstract

A real abstract kind of text

Acknowledgements

I would like to thank the entire DCModel group at ETH for making this thesis a thoroughly enjoyable and pleasant experience. Everyone has been exceedingly nice and helpful, and I felt comfortable and welcome at all times.

I am especially thankful to my supervisor, Dr. John Liagouris, for all the support he has given me throughout the thesis' work. He has been essential in keeping things on track, and in helping me resolve the various problems and hurdles that came up over time.

I would also like to specifically thank Dr. Frank McSherry for offering me a lot of his valuable time to explain and help discuss various aspects of the Timely system.

Finally I would like to thank Prof. Timothy Roscoe for allowing me to work on a thesis with a group of such excellent people.

Contents

1 Introduction **3**

2 Preliminaries **3**
2.1 Data Flow Essentials . . . 3
2.2 Timely Dataflow . . . 4

3 Related Work **4**
3.1 S4: Distributed stream computing platform . . . 5
3.2 SPADE: the system s declarative stream processing engine . . . 6
3.3 Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters . . . 6
3.4 MillWheel: fault-tolerant stream processing at internet scale . . . 7
3.5 Streamcloud: An elastic and scalable data streaming system . . . 7
3.6 Integrating scale out and fault tolerance in stream processing using operator state management . . . 8
3.7 Timestream: Reliable stream computation in the cloud . . . 8
3.8 Adaptive online scheduling in Storm . . . 9
3.9 Big data analytics on high Velocity streams: A case study . . . 10
3.10 Bigdatabench: A big data benchmark suite from internet services . . . 10
3.11 Comparison . . . 11

4 Benchmark Framework **13**
4.1 Benchmarks and Tests . . . 14
4.2 Input and Output System . . . 15
4.3 Additional Operators . . . 17
4.3.1 FilterMap . . . 17
4.3.2 Join . . . 18
4.3.3 Reduce . . . 21
4.3.4 RollingCount . . . 22
4.3.5 Window . . . 23
4.3.6 Session . . . 25
4.3.7 Partition . . . 27

5 Yahoo Streaming Benchmark (YSB) **27**
5.1 Implementation . . . 28
5.1.1 Data Generation . . . 28
5.1.2 Data Flow . . . 29
5.2 Evaluation . . . 30
5.3 Remarks . . . 31

6 HiBench: A Cross-Platforms Micro-Benchmark Suite for Big Data **32**
6.1 Implementation . . . 34
6.1.1 Data Generation . . . 34
6.1.2 Data Flows . . . 34

6.2 Evaluation 36
6.3 Remarks 38

7 NEXMark Benchmark 38
7.1 Implementation 43
7.1.1 Data Generation 43
7.1.2 Queries 43
7.2 Evaluation 49
7.3 Remarks 50

8 Conclusion 51

1 Introduction

2 Preliminaries

2.1 Data Flow Essentials

Many algorithms can be conveniently formulated as a data flow — a directed graph whose nodes represent operations performed on data, and whose edges represent the flow of data between such operators. This representation is especially handy for streaming systems, where the data is continuously fed into the system. Most of the streaming data processing systems in use today make use of data flows as the primary programming mechanism. Users express their programs in the form of several operators that are connected together, forming the *logical data flow*. The system then turns this logical data flow into a *physical data flow* by distributing the computation across *workers*. Workers represent physical processing units such as the cores of a processor, or individual machines in a network.

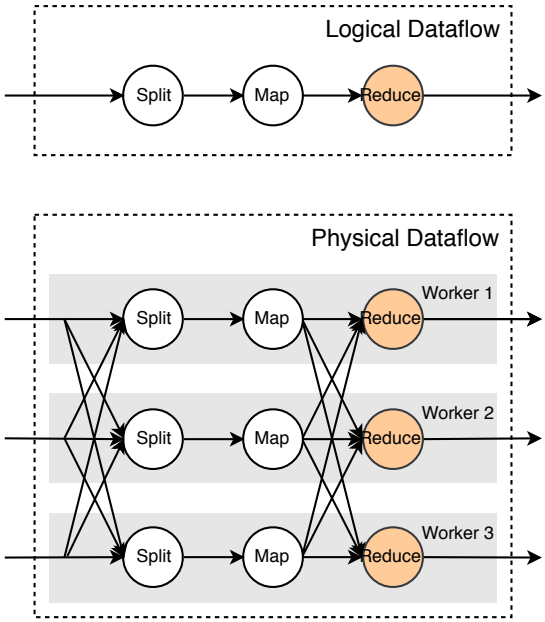

Figure 1: An illustration of the difference between logical (top) and physical (bottom) data flows. This illustration is based on the Timely model, where each operator specifies how the data it receives in the physical data flow should be redistributed between workers. The map operator for instance does not request redistribution of the data.

The data to be processed by each operator is distributed to each worker by the system, though the precise mechanisms of this vary with each system. Often the data is partitioned into disjoint sets, and each worker is assigned one set of this data.

After the worker has completed the operator's work, the generated output data is passed on to the next operator and might be redistributed across the workers.

2.2 Timely Dataflow

Timely[14] is a system written in Rust based on research initially proposed in Naiad[15]. Timely offers a dataflow based processing framework, where a dataflow is composed of a possibly cyclic graph of operators. Each operator can receive data from previous operators through input edges in the graph, and send data to other operators through output edges. Each edge represents a communication channel that is managed by the Timely system, which allows parallelising the computation to many workers at a time. Timely handles the exchange of data between operators and across workers.

Timely tracks progress of the dataflow computation through "epochs" — rounds of input data that are associated with a logical timestamp. These timestamps are required to implement a partial order, which is used to infer information about the progress of an operator. An important part of this inference is the notion of epoch closure: when an epoch is closed, no more input may arrive for it. Operators can request to be notified when specific epochs are closed, and can thus reason about when they have the full picture of the data. As a consequence of this, data before the closure of an epoch can arrive out of order, which typically improves performance as it lowers synchronisation constraints. The tracking of the epoch closure is called the "frontier".

The permission of cycles in the dataflow graph is achieved through "scopes". Any cycle must be encapsulated by such a scope. Each scope extends the timestamp by another dimension that tracks the progress within that scope. When the scope receives notification that an epoch has closed, it then continues processing until all inner epochs have been closed as well, at which point it can advance the frontier itself and propagate the information to the operators after the scope as well.

While the physical exchange of data is handled by Timely itself, data is only sent to other workers if requested, either by an explicit exchange operator, or by the communication contract specified by an operator. This allows the implementer of an operator to decide whether it makes sense to re-distribute the processing of the data. For instance, a keyed reduction would profit from having the data set partitioned over the workers according to the keys. A simple map on the other hand would not profit from having its data bucketed over the workers first.

3 Related Work

In this section we analyse and compare a number of papers about stream processors. In particular, we look at the ways in which they evaluate and test their systems in order to get an idea of how benchmarking has so far commonly been done. Each subsection looks at one paper at a time, providing a graph of the data flows and

operators used to evaluate the system, if such information was available. The operators are coloured in orange if they are stateful.

The papers were selected based on the number of citations, as well as on their direct relevance to current trends in the development and research for Big Data and streaming systems.

Overall we found that most of the systems were evaluated with relatively simple data flows and algorithms that are well understood. A lot of the papers also do not provide direct source code, nor a way to replicate the workload to confirm their findings. It seems that so far no generally accepted algorithm, workload, setup, nor even a precisely defined way of measuring performance have emerged.

3.1 S4: Distributed stream computing platform

The S4 paper[1] evaluates its performance with two algorithms: click-through rate (CTR), and online parameter optimisation (OPO).

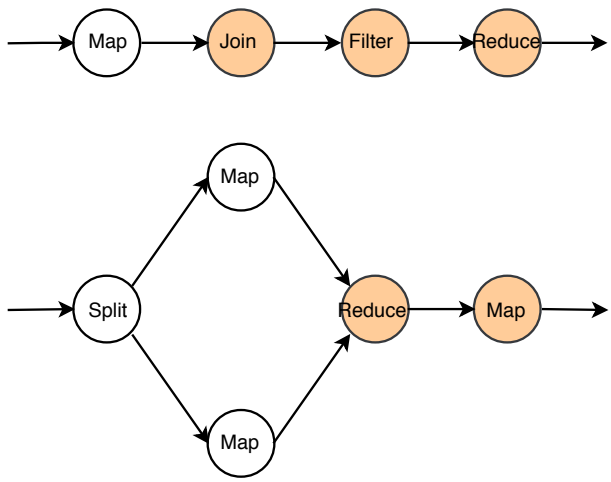

Figure 2: Graphs of the two data flows used to evaluate the S4 platform: click-through rate (top) and online parameter optimisation (bottom).

The CTR data flow is implemented by four operators: initially a map assigns key to the keyless events coming in. It passes them to a operator that combines matching events. From there the events go on to a filter that removes unwanted events. Finally, the events are passed to a operator that computes the CTR, and emits it as a new event.

The OPO data flow consists of five operators: the split operator assigns keys to route the events to either one of the map operators. These operators then perform

some computations on the events and emit the results as new events. The reduce operator compares the events it gets in order to determine the optimisation parameters. The final map operator runs an adaptation depending on the parameters it receives and passes them onwards.

3.2 SPADE: the system s declarative stream processing engine

In order to evaluate the system, the paper[2] employs a simple data flow to determine bargains to buy.

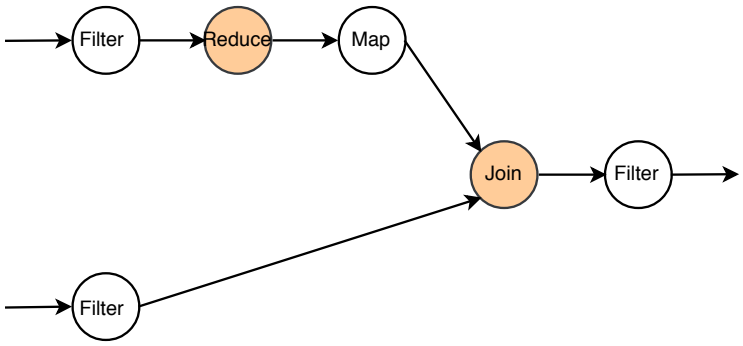

Figure 3: Graph of the example used in the SPADE paper: a bargain index computation.

The data flow is composed of six operators: a filter operator filters out trade information and computes its price. It passes its information on to a moving reduce operator, with a window size of 15 tuples, and a slide of 1 tuple. The reduction result is passed on to a mapping operator that computes the volume weighted average price (VWAP). Another filter operator filters out quote information from the main input stream. This is then, together with the VWAP, reduced to compute the bargain index. The final filter simply removes the zero indexes.

3.3 Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters

In the Discretized Streams paper[3], the performance is evaluated through a simple Word Count algorithm.

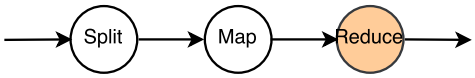

Figure 4: Graph of the Word Count example used to illustrate the discretized streams.

The Word Count data flow is implemented through three operators: a "flat map" that splits an incoming string into words, a map that turns each word into a tuple of the word and a counter, and finally a tumbling window reduction that adds the counters together grouped by word.

3.4 MillWheel: fault-tolerant stream processing at internet scale

The Millwheel paper[4] unfortunately provides barely any information at all about the data flows implemented. The only mention is about how many stages the pipelines have they use to evaluate the system. Two tests are performed: a single-stage data flow to measure the latency, and a three-stage data flow to measure the lag of their fault tolerance system.

3.5 Streamcloud: An elastic and scalable data streaming system

In this paper[5], the system is evaluated by two distinct data flows. It is not stated whether either of the data flows have any real-world application. The StreamCloud system provides a number of predefined operators that can be strung together to perform these data flows.

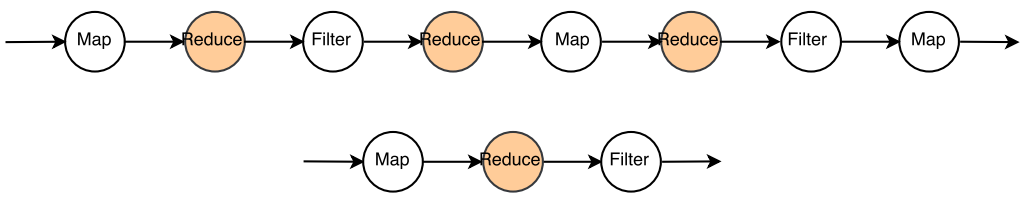

Figure 5: Graph of the data flows used to evaluate StreamCloud.

Both data flows perform a sequence of maps and filters followed by reductions. The reduction is based on a window size and slide, which can be configured for each operator. However, the configurations used are not provided by the paper.

3.6 Integrating scale out and fault tolerance in stream processing using operator state management

To evaluate their approach for fault tolerance using Operator State Management[6], two data flows were implemented: a linear road benchmark (LRB) to determine tolls in a network, and a Top-K data flow to determine the top visited pages. The data flow is composed out of stateless and stateful operators, where stateful operators must communicate their state to the system so that it may be recovered.

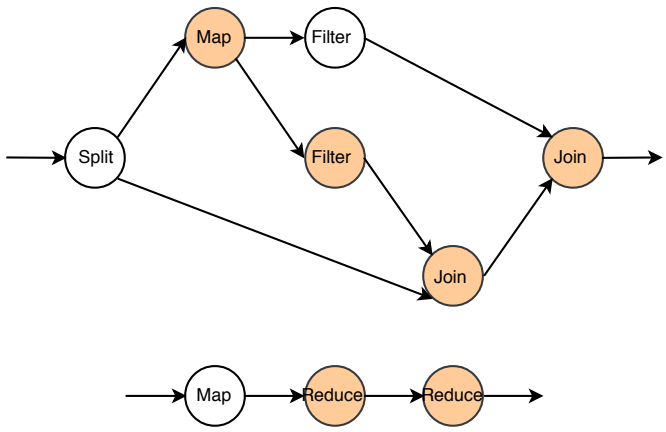

Figure 6: Illustration of the data flows for the Linear Road Benchmark (top) and Top-K (bottom) used to evaluate their system.

The LRB is implemented using six operators. The first split operator routes the tuples depending on their type. The following map operator calculates tolls and accidents, the information of which is then forwarded to a operator that collects toll information, and a operator that evaluates the toll information. The output from the evaluation, together with account balance information, is aggregated and finally reduced to a single tuple together with the information from the toll collector operator.

The Top-K data flow is implemented using three operators. The starting map operator strips unnecessary information from the tuples. The following operator reduces the tuples to local top-k counts. Finally the many local counts are reduced to a single top-k count for the whole data.

3.7 Timestream: Reliable stream computation in the cloud

The TimeStream system[7] is evaluated using two algorithms: a distinct count to count URLs and a Twitter sentiment analysis.

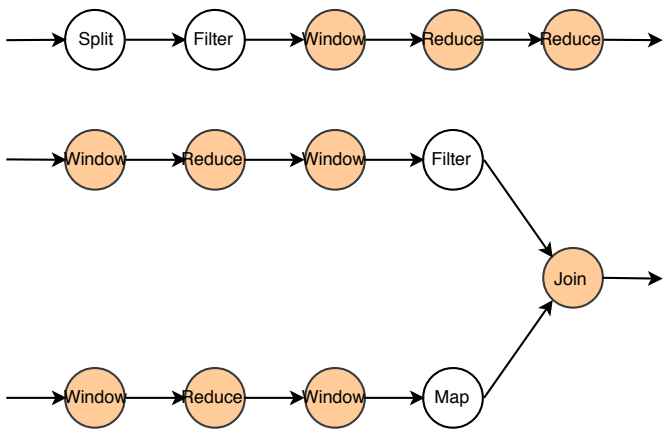

Figure 7: The Distinct Count (top) and Sentiment Analysis (bottom) data flows used to evaluate the Timestream system.

The distinct count is implemented using five operators. The split operator distributes the tuples based on a hash. The following filter removes bot-generated data flows, and passes them on to a windowing operator with a window of 30 seconds and a slide of 2 seconds. The windowed events are then reduced into local counts. The local counts are finally reduced into global counts.

The sentiment analysis performs two individual computations before finally joining the results together with a custom operator. The first computation determines changes in sentiments. It uses a tumbling window on the tweets, averages the sentiments, for each window, then uses a sliding window of 2 ms to feed a filter that only returns sentiments that changed. The second computation returns the change in word counts. It uses a tumbling window of the same size as the first computation, then reduces the word counts for each batch. Using another sliding window of 2 ms it then computes a delta in the counts. Using a custom operator the sentiment changes and word count deltas are then joined together to analyse them.

3.8 Adaptive online scheduling in Storm

This paper proposes a new scheduling algorithm for Storm. It then uses a data flow specifically geared towards evaluating the scheduling. This data flow is believed to be representative of typical topologies found in applications of Storm.

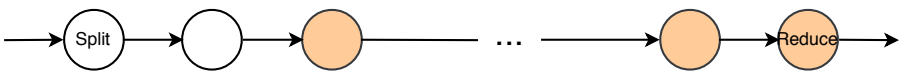

Figure 8: The topology graph used to evaluate the Storm schedulers.

The data flow is composed of a sequence of operators that produce arbitrary, new events distinguished by a counter. The data flow has no interesting properties aside from the alternation between stateless and stateful operators.

3.9 Big data analytics on high Velocity streams: A case study

This paper[9] presents a case study to perform real-time analysis of trends on Twitter using Storm.

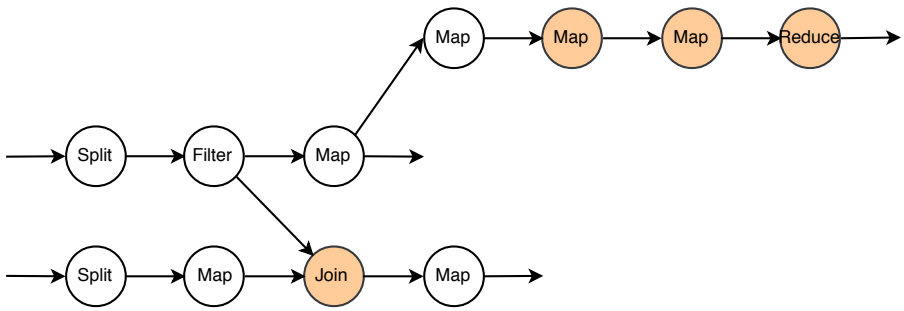

Figure 9: An illustration of the topology used for the Twitter & Bitly link trend analysis.

The data flow for this is the most complicated one presented in the related works we analysed. It uses a total of eleven operators, excluding edge operators that act as interfaces to the external systems. The computation can be separated into three stages: Twitter extraction, Bit.ly extraction, and trend analysis. The first stage filters out tweets that contain Bit.ly links. Those are then sent to the Bit.ly extraction stage and a map that extracts useful values for the trend analysis. The second stage extracts relevant information from the Bit.ly feed, then uses this together with the code received from the first stage to perform a Bloom filter. The output from there is then filtered for useful values before being saved. The trend analysis uses the extracted values from the tweets to find hashtags, which are then put through a rolling-window count. The resulting counts are reduced by two stages of ranking.

3.10 Bigdatabench: A big data benchmark suite from internet services

This paper[10] proposes a suite of benchmarks and tests to evaluate Big Data systems. The paper primarily focuses on the generation of suitable testing data sets, and proposes the following algorithms to test the system:

- Sort
- Grep
- Word Count
- Retrieving Data
- Storing Data
- Scanning Data
- Select Query
- Aggregate Query
- Join Query
- Nutch Server
- Indexing
- Page Rank
- Olio Server
- K-means
- Connected Components
- Rubis Server
- Collaborative Filtering
- Naive Bayes

The paper does not propose any particular implementation strategies. They provide performance evaluation for an implementation of different parts of the benchmark suite on the Hadoop, MPI, Hbase, Hive, and MySQL systems, but no particular details of the implementation are discussed.

3.11 Comparison

In Table 1 and Table 2 we compare the most important features of the tests performed in the various papers. Unfortunately, most of the papers do not supply or use publicly available data, making it difficult to compare them, even if the data flows were replicated.

Paper	**Goal**	**Application**	**Data Flow Properties**	**Data Flow Operators**
S4[1]	A practical application of the system to a real-life problem.	Search	Stateful, DAG	Map, Filter, Join
SPADE[2]	Sample application, performance study.	Finance	Stateful, DAG	Map, Filter, Window Reduce, Join
D-Streams[3]	Scalability and recovery test.	None	Chain	Map, Window Reduce
Millwheel[4]	In-Out Latency.	Ads	Unspecified	Unspecified
StreamCloud[5]	Evaluation of scalability and elasticity.	Telephony	Stateful, Chain	Map, Filter, Window Reduce
Seep[6]	Testing dynamic scaling and fault-tolerance.	Road tolls	Stateful, DAG	Map, Reduce, Join
TimeStream[7]	Low-latency test for real-world applications.	Search, Social Network	DAG	Map, Filter, Reduce, Join, Window
Adaptive Scheduling[8]	Evaluating performance of scheduling algorithms.	None	Stateful, Chain	Reduce
Analytics on High Velocity Streams[9]	Analysing trends for links on Twitter.	Social Network	Stateful, DAG	Map, Filter, Reduce, Join, Window
BigDataBench[10]	Fair performance evaluation of big data systems.	Search, Social, Commerce	Unspecified	Unspecified
YSB[11]	Benchmarking streaming systems via Ad analytics.	Ads	Stateful	Map, Filter, Reduce, Join, Window
HiBench[12]	Evaluating big data processing systems.	Big Data	Stateful	Map, Reduce, Window
NEXMark[13]	Adaptation of XMark for streaming systems.	Auctioning	Stateful	Map, Filter, Reduce, Join, Window, Session

Table 1: Comparison of the test properties of the reference papers.

Paper	Workloads	Testbed	External Systems	Public Data
S4[1]	~1M live events per day for two weeks.	16 servers with 4x32-bit CPUs, 2GB RAM each.	Unspecified	No
SPADE[2]	~250M transactions, resulting in about 20GB of data.	16 cluster operators. Further details not available.	IBM GPFS	Maybe[1]
D-Streams[3]	~20 MB/s/operator (200K records/s/operator) for WordCount.	Up to 60 Amazon EC2 operators, 4 cores, 15GB RAM each.	Unspecified	No
Millwheel[4]	Unspecified.	200 CPUs. Nothing further is specified.	BigTable	No
StreamCloud[5]	Up to 450'000 transactions per second.	100 operators, 32 cores, 8GB RAM, 0.5TB disks each, 1Gbit LAN.	Unspecified	No
Seep[6]	Up to 600'000 tuples/s.	Up to 50 Amazon EC2 "small" instances with 1.7GB RAM.	Unspecified	No
TimeStream[7]	~30M URLs, ~1.2B Tweets.	Up to 16 Dual Xeon X3360 2.83GHz, 8GB RAM, 2TB disks each, 1Gbit LAN.	Unspecified	No
Adaptive Scheduling[8]	Generated.	8 operators, 2x2.8GHz CPU, 3GB RAM, 15GB disks each, 10Gbit LAN.	Nimbus, Zookeper	Yes[2]
Analytics on High Velocity Streams[9]	~1'600GB of compressed text data.	4 operators, Intel i7-2600 CPU, 8GB RAM each.	Kafka, Cassandra	Maybe[3]
BigDataBench[10]	Up to 1TB.	15 operators, Xeon E5645, 16GB RAM, 8TB disks each.	Hadoop, MPI, Hbase, Hive, MySQL	Yes[4]
YSB[11]	Generated	Unspecified	Kafka, Redis	Yes[5]
HiBench[12]	Generated	Unspecified	Kafka	Yes[6]
NEXMark[13]	Generated	Unspecified	Firehose Stream Generator	Yes[7]

Table 2: Comparison of the test setups of the reference papers.

4 Benchmark Framework

In order to factor out common operations and to be able to cleanly define the data flows of each test in the benchmarks we developed a new framework. This framework takes care of processing command line arguments, Timely setup, and statistical data collection. All that's necessary in order to implement a benchmark is the implementation of the `Benchmark` trait for the overall benchmark, the implementation of the `TestImpl` trait for each test in the benchmark, and the implementation of necessary data conversion routines.

[1] The data was retrieved from the IBM WebSphere Web Front Office for all of December 2005.
[2] The data is generated on the fly, the algorithm of which is specified in the paper.
[3] Data stems from Twitter and Bit.ly for June of 2012, but is not publicly available.
[4] Obtainable at http://prof.ict.ac.cn/BigDataBench/
[5] Generated by YSB: https://github.com/yahoo/streaming-benchmarks
[6] Generated by HiBench.
[7] Generated by the "Firehose Stream Generator".

4.1 Benchmarks and Tests

```
pub trait Benchmark {
    fn name(&self) -> &str;
    fn generate_data(&self, config: &Config) -> Result<()>;
    fn tests(&self) -> Vec<Box<Test>>;
}
```

Listing 1: Definition of the Benchmark trait.

The `Benchmark` includes a method to generate input data files that can be used to feed into the system from disk at a later point, and a function to return all the tests that the benchmark defines. Note that it returns `Test` trait instances rather than `TestImpl`. This is done to avoid leaking associated trait types into surrounding code that doesn't need to know about it.

```
pub trait Test : Sync+Send {
    fn name(&self) -> &str;
    fn run(&self, config: &Config, worker: &mut Root<Generic>)
        -> Result<Statistics>;
}
```

Listing 2: Definition of the Test trait.

The `Test` trait includes only one important method, which runs the test on a given worker and returns collected statistical data of the run if it completed was successfully. The `Test` trait is automatically implemented for all types that implement the `TestImpl` trait, thus avoiding the need to manually implement the `Test` trait, and automatically erasing the associated types required in the `TestImpl` definition.

```
pub trait TestImpl : Sync+Send {
    type D: Data;
    type DO: Data;
    type T: Timestamp;

    fn name(&self) -> &str;
    fn create_endpoints(&self, &Config, index: usize, workers: usize)
        -> Result<(Source<Self::T, Self::D>, Drain<Self::T, Self::DO>)>;
    fn construct_dataflow<'scope>(&self, &Config, stream:
          &Stream<Child<'scope, Root<Generic>, Self::T>, Self::D>)
        -> Stream<Child<'scope, Root<Generic>, Self::T>, Self::DO>;
    fn run(&self, &Config, worker: &mut Root<Generic>)
        -> Result<Statistics>;
}
```

Listing 3: Definition of the TestImpl trait.

The `TestImpl` trait is the most complicated, but should not be hard to utilise either. The `run` method has a default implementation and as such typically does

not need to be implemented. This implementation makes use of the associated `D`, `DO`, and `T` types in order to work regardless of the test's type requirements. The `create_endpoints` method is responsible for initialising the input and output of the data flow. This method is required to be implemented in order for the test to function properly, but should be very straight-forward.

```
fn create_endpoints(&self, config: &Config, _: usize, _: usize)
   -> Result<(Source<Self::T, Self::D>, Drain<Self::T, Self::DO>)> {
    Ok((Source::from_config(config,
          Source::new(Box::new(NEXMarkGenerator::new(config))))?,
        Drain::from_config(config)?))
}
```

Listing 4: A sample definition of the `create_endpoints` method.

Finally, the most important method is `construct_dataflow`. It is responsible for transforming the input data stream into an output data stream, implementing the data flow logic. The function bodies of the respective `construct_dataflow` implementations are shown in the Data Flows subsections of each benchmark. A minimal complete example of a test definition is shown in listing 5.

```
struct MyTest{}

impl TestImpl for MyTest{
    type T = usize;
    type D = String;
    type DO = (String, usize);

    fn create_endpoints(&self, _:&Config, _: usize, _: usize)
        -> Result<(Source<Self::T, Self::D>, Drain<Self::T, Self::DO>)>{
        Ok(Source::from(vec!((0, vec!("a", "b").map(String::from)),
                             (1, vec!("b", "c").map(String::from)))),
           Drain::from(io::stdout()))
    }

    fn construct_dataflow<'scope>(&self, _:&Config, stream:
          &Stream<Child<'scope, Root<Generic>, Self::T>, Self::D>)
        -> Stream<Child<'scope, Root<Generic>, Self::T>, Self::DO>{
        stream.rolling_count(|w| w.clone(), |w, c| (w, c))
    }
}
```

Listing 5: A complete sample test definition. The endpoints are configured to feed data from a predefined vector, and output results to the standard output stream. The dataflow performs a simple word count.

4.2 Input and Output System

In order to factor out the common problem of handling the data input and result output of the dataflows, we developed a simple "endpoints" system. At the fore-

front are the opaque `Source` and `Drain` instances that wrap an inner `EventSource` and `EventDrain` trait instance respectively. This allows us to pass the sources and drains around without the rest of the system having to know about their particular properties.

```
pub struct Source<T, D>(Box<EventSource<T, D>>);
pub struct Drain<T, D>(Box<EventDrain<T, D>>);

pub trait EventSource<T, D> {
    fn next(&mut self) -> Result<(T, Vec<D>)>;
}

pub trait EventDrain<T, D> {
    fn next(&mut self, T, Vec<D>);
}
```

Listing 6: The core traits and structs of the endpoint system.

We provide the following implementations of the `EventSource` and `EventDrain` traits for ease of use:

- `Null` — This struct does not provide any data as `EventSource`, and simply ignores all data as `EventDrain`.
- `Console` — As `EventSource`, this struct reads from the standard input stream and converts each line into input data. As `EventDrain` it simply writes each timestamp and data pair to the standard output.
- `FileInput` — This `EventSource` reads lines from a file and, similar to the `Console`, converts each line into a single input record.
- `FileOutput` — This `EventDrain` writes each record it is given to a file, one line per record.
- `VectorEndpoint` — This can be used both as `EventSource` and `EventDrain`, providing an in-memory stream for records.
- `MeterOutput` — This `EventDrain` simply reports how many records it sees for each epoch.

The `From` trait is implemented on both `Source` and `Drain` wherever applicable, such that constructing the proper endpoints is relatively trivial.

```
pub struct Null{}

impl<T, D> EventSource<T, D> for Null {
    fn next(&mut self) -> Result<(T, Vec<D>)> {
        out_of_data()
    }
}

impl<T: Timestamp, D: Data> From<()> for Source<T, D> {
    fn from(_: ()) -> Source<T, D> {
        Source::new(Box::new(Null{}))
    }
}
```

Listing 7: The definition of the null event source.

Some of these endpoints require the conversion of data to and from a serialised format such as strings. In order to facilitate this, the `ToData` and `FromData` traits need to be implemented by the user to perform the required conversion between their data types. An example of such conversion routines for the code from listing 5 is provided in listing 8.

```
impl FromData<usize> for (String, usize) {
    fn from_data(&self, t: &usize) -> String {
        format!("{} {} ×{}", t, self.0, self.1)
    }
}
```

Listing 8: Record conversion to make MyTest work.

4.3 Additional Operators

In order to ease the implementation of the benchmarks and improve the usability of the Timely system we introduced a number of additional operators. We will outline and discuss these operators here shortly.

4.3.1 FilterMap

This operator performs a filter followed by a map in one go. The operator expects a closure which can choose to either filter events by returning `None`, or map them by returning `Some(..)` with the output data.

```
fn filter_map(&self, map) -> Stream {
    self.unary_stream(move |input, output| {
        input.for_each(|time, data| {
            let mut session = output.session(time);
            data.for_each(|x|{
                if let Some(d) = map(x) {
                    session.give(d);
                }
            });
        });
    })
}
```

Listing 9: Simplified code for the filter map operator.

The operator is mostly useful for streams that contain multiple types of data encapsulated in an enum. In that case turning the stream into one of a single type is trivial using `filter_map`.

```
enum Vehicle {
    Car(Car)
    Boat(Boat)
}

impl Car {
    fn from(vehicle: Vehicle) -> Option<Car> {
        match vehicle {
            Vehicle::Car(car) => Some(car),
            _ => None
        }
    }
}

stream.filter_map(|x| Car::from(x))
```

Listing 10: An example of the filter map operator to purify a stream of vehicles into one of cars.

4.3.2 Join

This operator offers two forms of joins that merge two separate streams of data into one. The first is an epoch based join, meaning data is only matched up between the two streams within a single epoch. If no match is found for either stream, the data is discarded.

```
fn epoch_join(&self, stream, key_1, key_2, joiner) -> Stream{
    let mut epoch1 = HashMap::new();
    let mut epoch2 = HashMap::new();

    self.binary_notify(move |input1, input2, output, notificator| {
        input1.for_each(|time, data|{
            let epoch = epoch1.entry(time).or_insert_with(HashMap::new);
            data.for_each(|dat|{
                let key = key_1(&dat);
                let datavec = epoch.entry(key).or_insert_with(Vec::new);
                datavec.push(dat);
            });
            notificator.notify_at(time);
        });

        input2.for_each(|time, data|{
            let epoch = epoch2.entry(time).or_insert_with(HashMap::new);
            data.for_each(|dat|{
                let key = key_2(&dat);
                let datavec = epoch.entry(key).or_insert_with(Vec::new);
                datavec.push(dat);
            });
            notificator.notify_at(time);
        });

        notificator.for_each(|time, _, _|{
            if let Some(k1) = epoch1.remove(time) {
                if let Some(mut k2) = epoch2.remove(time) {
                    let mut out = output.session(time);
                    for (key, data1) in k1{
                        if let Some(mut data2) = k2.remove(&key) {
                            for d1 in data1 {
                                data2.for_each(|d2|
                                    out.give(joiner(d1.clone(), d2)));
                            }
                        }
                    }
                }
            } else {
                epoch2.remove(time);
            }
        });
    })
}
```

Listing 11: Simplified code for the epoch based join operator.

The epoch join is the more general join operator that should be useful whenever it is ensured that related events are emitted in the same epoch. If the out-of-orderness means the epochs could be different, a more general join operator is required.

```
customers.epoch_join(coffees,
    |customer| customer.id,
    |coffee| coffee.customer,
    |customer, coffee| (customer.name, coffee.price))
```

Listing 12: An example of an epoch join to determine how much each customer needs to pay for their coffee.

The second form of join offered is a left join that keeps the left-hand stream's data around indefinitely, continuously joining it with data from the right-hand stream whenever the keys match.

```
fn left_join(&self, stream, key_1, key_2, joiner) -> Stream{
    let mut d1s = HashMap::new();
    let mut d2s = HashMap::new();

    self.binary_notify(stream, move |input1, input2, output, _| {
        input1.for_each(|time, data|{
            data.for_each(|d1| {
                let k1 = key_1(&d1);
                if let Some(mut d2) = d2s.remove(&k1) {
                    output.session(time).give_iterator(
                        d2.map(|d| joiner(d1.clone(), d)));
                }
                d1s.insert(k1, d1);
            });
        });
        input2.for_each(|time, data|{
            data.for_each(|d2| {
                let k2 = key_2(&d2);
                if let Some(d1) = d1s.get(&k2) {
                    output.session(time).give(joiner(d1.clone(), d2));
                } else {
                    d2s.entry(k2).or_insert_with(Vec::new).push(d2);
                }
            });
        });
    })
}
```

Listing 13: Simplified code for the left join operator.

Left joins are useful when events on one side might only be emitted once, but events on the right hand side are recurring, meaning we need to retain the left side indefinitely.

```
driver_registrations.left_join(speeding_cars,
    |car| car.driver,
    |driver| driver.id,
    |car, driver| (car.license_plate, driver.address))
```

Listing 14: An example of the left join operator, joining driver's registrations to cars that have been caught speeding.

4.3.3 Reduce

Reducing data in some form is a very frequent operation in dataflows. This operator offers multiple variants of reduction for ease-of-use. A generic `reduce` that requires a key extractor, an initial value, a reductor, and a completor. The key extractor decides the grouping of the data, and the reductor is responsible for computing the intermediate reduction result for every record that arrives. Once an epoch is complete, the completor is invoked in order to compute the final output data from the intermediate reduction, the count of records, and the key for this batch of records. The variants `reduce_by`, `average_by`, `maximize_by`, and `minimize_by` build on top of this to provide more convenient access to reduction.

```
fn reduce(&self, key_extractor, initial, reductor, completor) -> Stream{
    let mut epochs = HashMap::new();

    self.unary_notify(move |input, output, notificator| {
        input.for_each(|time, data| {
            let window = epochs.entry(time).or_insert_with(HashMap::new);
            data.for_each(|dat|{
                let key = key(&dat);
                let (v, c) = window.remove(&key).or((initial, 0));
                let value = reductor(dat, v);
                window.insert(key, (value, c+1));
            });
            notificator.notify_at(time);
        });
        notificator.for_each(|time, _, _| {
            if let Some(mut window) = epochs.remove(time) {
                output.session(time).give_iterator(
                    window.map(|(k, (v, c))| completor(k, v, c)));
            }
        });
    })
}
```

Listing 15: Simplified code for the general reduce operator.

```
products.reduce(|_| 0, Product::new(0), |product, highest| {
    if highest.price < product.price { product } else { highest }
})
```

Listing 16: A reduction example to find the product with the highest price.

Finally, a separate `reduce_to` operator does not key data and instead reduces all data within the epoch to a single record.

```
fn reduce_to(&self, initial_value, reductor) -> Stream {
    let mut epochs = HashMap::new();

    self.unary_notify(move |input, output, notificator| {
        input.for_each(|time, data| {
            let mut reduced = epochs.remove(time).or(initial_value);
            while let Some(dat) = data.pop() {
                reduced = reductor(dat, reduced);
            };
            epochs.insert(time, reduced);
            notificator.notify_at(time);
        });
        notificator.for_each(|time, _, _| {
            if let Some(reduced) = epochs.remove(time) {
                output.session(time).give(reduced);
            }
        });
    })
}
```

Listing 17: Simplified code for the reduce to operator.

This operator can be useful when trying to compare against a common value among all records.

```
records.reduce_to(0, |_, c| c+1)
```

Listing 18: An example showing how to count the number of records in an epoch.

4.3.4 RollingCount

The `rolling_count` operator is similar to a reductor, but has a few distinct differences. First, it emits an output record for every input record it sees, rather than only once per epoch. Second, it keeps the count across epochs, rather than resetting for each epoch. Finally, it can only count records, rather than performing arbitrary reduction operations.

```
fn rolling_count(&self, key_extractor, counter) -> Stream{
    let mut counts = HashMap::new();

    self.unary_stream(move |input, output| {
        input.for_each(|time, data| {
            output.session(time).give_iterator(data.map(|x|{
                let key = key(&x);
                let count = counts.get(&key).unwrap_or(0)+1;
                counts.insert(key.clone(), count);
                counter(x, count)
            }));
        });
    })
}
```

Listing 19: Simplified code for the rolling count operator.

The most trivial use-case is the classic word count benchmark.

```
words.rolling_count(|word| word.clone(), |word, count| (word, count))
```

Listing 20: A basic word count example using the rolling-count operator.

4.3.5 Window

The window operator batches records together into windows. Windows can be sliding or hopping, and can be of arbitrary size, although they are limited in their granularity by epochs. This means that the epochs need to be correlated to a unit that the user would like to window by. When the window is full and the frontier reaches a slide, the window operator sends out a copy of all records within the window.

```
fn window(&self, size, slide, time) -> Stream
    let mut parts = HashMap::new();
    self.unary_notify(move |input, output, notificator| {
        input.for_each(|cap, data| {
            data.drain(..).for_each(|data|{
                let time = time(cap.time(), &data);
                // Push the data onto a partial window.
                let part = parts.entry(time).or_insert_with(Vec::new);
                part.push(data);
                // Calculate the next epochs on which this partial window
                // would be output in a slide, then notify on those times
                for i in 0..size/slide {
                    let target = if time < size { size-1
                    } else { size-1+((time-size)/slide+1+i)*slide };
                    notificator.notify_at(cap.delayed(target));
                }
            });
        });

        notificator.for_each(|cap, _, _| {
            let end = cap.time();
            let mut time = end+1-size;
            let slide_end = time+slide;
            let mut window = Vec::new();
            // Compute full window from partials. First gather parts
            // that would fall out of the window and remove them.
            while time < slide_end {
                if let Some(mut part) = parts.remove(time) {
                    window.append(&mut part);
                }
                time += 1;
            }
            // Then gather and clone parts that will still be relevant
            // later.
            while time <= end {
                if let Some(part) = parts.get(time) {
                    part.iter().for_each(|e| window.push(e.clone()));
                }
                time += 1;
            }
            // Finally output the full window.
            output.session(&cap).give_iterator(window.drain(..));
        });
    })
}
```

Listing 21: Simplified code for the general window operator.

The generic window operator has some overhead. Thus there is a specific operator for tumbling windows which should be a lot more efficient.

```
fn tumbling_window(&self, size) -> Stream{
    let mut windows = HashMap::new();

    self.unary_notify(move |input, output, notificator| {
        let size = size.clone();
        input.for_each(|cap, data| {
            // Round the time up to the next window.
            let wtime = (cap.time() / size + 1) * size;
            // Now act as if we were on that window's time.
            notificator.notify_at(cap.delayed(wtime));
            let window = windows.entry(wtime).or_insert_with(Vec::new);
            data.drain(..).for_each(|data|{
                window.push(data);
            });
        });

        notificator.for_each(|cap, _, _| {
            if let Some(mut window) = windows.remove(cap.time()) {
                output.session(&cap).give_iterator(window.drain(..));
            }
        });
    })
}
```

Listing 22: Simplified code for the tumbling window operator.

A typical example of a tumbling window usage is to batch events together into well-defined intervals.

```
frames.tumbling_window(60)
```

Listing 23: An example of a tumbling window, batching frames into intervals of minutes, assuming an epoch represents one second.

4.3.6 Session

The session operator is similar to a window: it batches records, but instead of using a regular interval, a session is only completed after a certain timeout has been reached. As an example, a session with a timeout of 10 seconds would only be complete if there were no records for 10 seconds on the stream. Before this timeout is reached, all incoming records are gathered into the current session.

```
fn session(&self, timeout, sessioner) -> Stream{
    let mut sessions = HashMap::new();

    self.unary_notify(move |input, output, notificator| {
        input.for_each(|cap, data| {
            for data in data.drain(..){
                let (s, t) = key(&data);
                notificator.notify_at(cap.delayed(t + timeout));
                let session = sessions
                    .entry(t).or_insert_with(HashMap::new)
                    .entry(s).or_insert_with(Vec::new);
                session.push(data);
            };
        });

        notificator.for_each(|cap, _, _| {
            // For each session at the original time we need to check if
            // it has expired, or if we need to delay.
            let otime = cap.time() - timeout;
            let mut expired = sessions.remove(&otime).or(HashMap::new);
            expired.drain().for_each(|(s, mut d)| {
                // Now we check backwards from the current epoch.
                let mut found = false;
                for i in 0..timeout {
                    let t = cap.time() - i;
                    if let Some(session) = sessions.get_mut(&t) {
                        if let Some(data) = session.get_mut(&s) {
                            // If we find data within the timeout, delay
                            // our data to that later time. If that time
                            // does not happen to be final either, both
                            //  this and that data will get moved ahead
                            //even further automatically.
                            data.append(&mut d);
                            found = true;
                            break;
                        }
                    }
                }
                if !found {
                    // If we don't find a any data within the timeout,
                    // the session is full and we can output it.
                    output.session(&cap).give((s, d));
                }
            });
        });
    })
}
```

Listing 24: Simplified code for the session operator.

Sessions can be useful to track intervals of activity, for instance to try and estimate periods of time during which a user is actively visiting a site.

```
tweets.session(3600, |tweet| (tweet.author, tweet.time / 1000))
```

Listing 25: An example of a session to determine batches of tweets during which the user is active.

4.3.7 Partition

The partitioning operator transforms the data stream into windows of a fixed number of records, each keyed by a property.

```
fn partition(&self, size, key) -> Stream{
    let mut partitions = HashMap::new();

    self.unary_stream(move |input, output| {
        input.for_each(|time, data| {
            data.for_each(|dat| {
                let key = key(&dat);
                let mut partition = partitions.remove(&key)
                                        .or(|| Vec::with_capacity(size));
                partition.push(dat);
                if partition.len() == size {
                    output.session(time).give(partition);
                } else {
                    partitions.insert(key, partition);
                }
            });
        });
    })
}
```

Listing 26: Simplified code for the partitioning operator.

Partitioning is useful when we need to be certain about the number of events present in the stream at any particular point.

```
frames.partition(30, |frame| frame.animation)
```

Listing 27: This creates 30 frame (one second) animation batches from a stream of frames.

5 Yahoo Streaming Benchmark (YSB)

The Yahoo Streaming Benchmark[11] is a single dataflow benchmark created by Yahoo in 2015. Of the three benchmark suites implemented in this thesis, it is the one most widely used in the industry. The original implementation includes support for Storm, Spark, Flink, and Apex. The benchmark only focuses on the latency aspect

of a streaming system, ignoring other important factors such as scaling, fault tolerance, and load bearing.

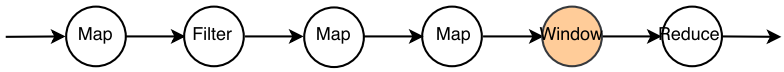

Figure 10: A graph of the dataflow described by YSB.

The dataflow used in the benchmark is illustrated in Figure 10. Its purpose is to count ad hits for each ad campaign. Events arrive from Kafka in JSON string format, where each event is a flat object with the following fields:

- `user_id` A UUID identifying the user that caused the event.
- `page_id` A UUID identifying the page on which the event occurred.
- `ad_id` A UUID for the specific advertisement that was interacted with.
- `ad_type` A string, one of "banner", "modal", "sponsored-search", "mail", and "mobile".
- `event_type` A string, one of "view", "click", and "purchase".
- `event_time` An integer timestamp in milliseconds of the time the event occurred.
- `ip_address` A string of the user's IP address.

The dataflow proceeds as follows: the first operator parses the JSON string into an internal object. Irrelevant events are then filtered out, and only ones with an `event_type` of "view" are retained. Next, all fields except for `ad_id` and `event_time` are dropped. Then, a lookup in a table mapping `ad_id`s to `campaign_id`s is done to retrieve the relevant `campaign_id`. Yahoo describes this step as a join, which is inaccurate, as only one end of this "join" is streamed, whereas the other is present as a table stored in Redis. Next the events are put through a ten seconds large hopping window. The number of occurrences of each `campaign_id` within each window are finally counted and stored back into Redis.

5.1 Implementation

5.1.1 Data Generation

As the data used in YSB is fairly straight forward, we created our own data generator. The generator creates random `user_id`s, `page_id`s, and `ip_address`es. Since those fields aren't actually touched by the query, the precise data should not make any difference. The `ad_type` and `event_type` are randomly chosen from the specified

sets. The `ad_id` is chosen from a randomly generated table of `ad_id`s to `campaign_id`s. This table consists of 100 campaigns with 10 ads each, as specified by YSB. The most interesting field is the `event_time` which is monotonically stepped in milliseconds according to how many events per second should be generated.

This is all in line with the implementation of the data generator found in the original YSB repository (`data/src/setup/core.clj`). Curiously, their implementation does include parts to skew the time stamps and randomise them, but they are not actually used. Like many other implementations of YSB, we also do not rely on Redis for the `ad_id` to `campaign_id` lookup, and instead keep this small table in memory.

The primary purpose of implementing our own generation for YSB is to find a short path to testing the query. While it is possible to add Kafka as an event source, generating the data directly in memory allows us to more easily test various configurations and explore datasets that would take up massive amounts of space to store ahead of time.

5.1.2 Data Flow

```
let table = self.campaign_map.read().unwrap().clone();
stream
    .filter(|x: &Event| x.event_type == "view")
    .map(|x| (x.ad_id, x.event_time))
    .map(move |(ad_id, _)|
         match table.get(&ad_id){
             Some(id) => id.clone(),
             None => String::from("UNKNOWN AD")
         })
    .tumbling_window(window_size)
    .reduce_by(|campaign_id| campaign_id.clone(), 0, |_, count| count+1)
```

Listing 28: Dataflow implementation of the YSB benchmark.

The implementation of the query is rather straightforward. The only step of the dataflow graph not directly represented here as an operator is the translation of the JSON string into the event object. We skip out on this as the translation is done on the data feeding end in order to use the event's `event_time` field to manage the corresponding epoch of the event. Each epoch corresponds to a real-time of one second.

The second `map` operator is responsible for performing the lookup in the `campaign_id` table. Instead of the original Redis query, we use a simple hash-table lookup. A copy of the table is kept locally in memory of each worker. Since we don't make use of any of the remaining event fields after this step, we only emit the `campaign_id`, rather than a tuple of the event's fields.

5.2 Evaluation

We ran our experiments on an AMD Opteron 6378 2.4KHz 64bit machine with a total of 32 Cores and 504GB RAM. This machine is known to exhibit strange scaling behaviour due to non-uniform memory access patterns. You can see this behaviour in the scaling plot at 16 workers.

Our measurement procedure involved a closed-loop data feed, meaning each epoch was run to completion before a new epoch with a new round of data was started. We only measured data flow execution time, excluding data generation time. Each measurement was gathered using 300 epochs of data.

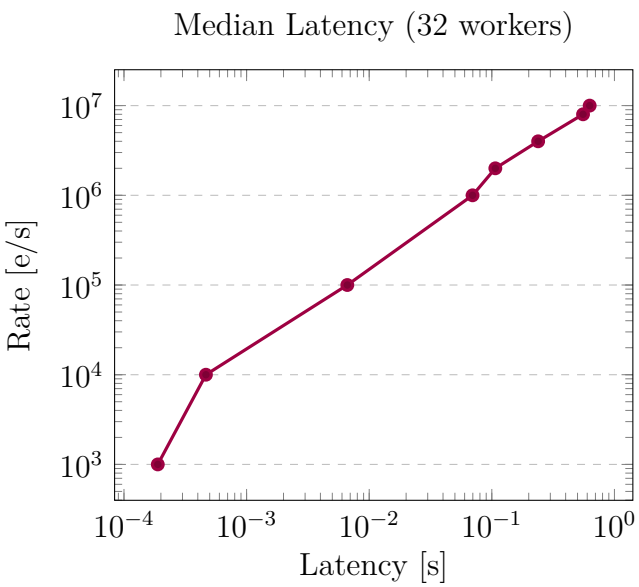

Figure 11: Loglog plot of the Latency vs Rate.

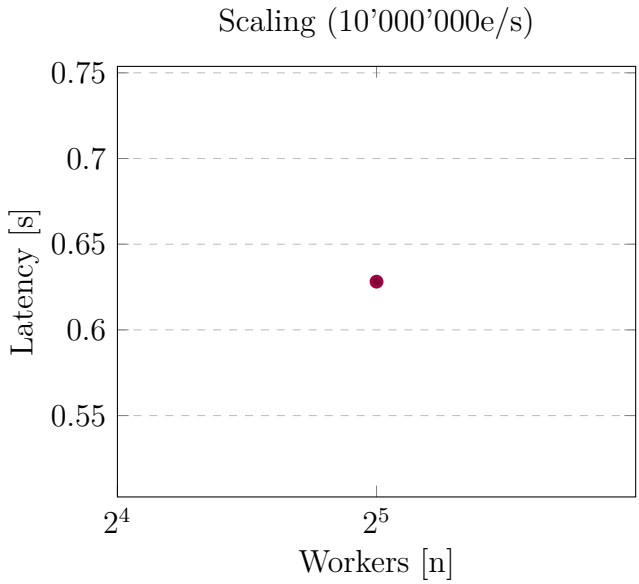

Figure 12: Linlog plot of the scaling behaviour.

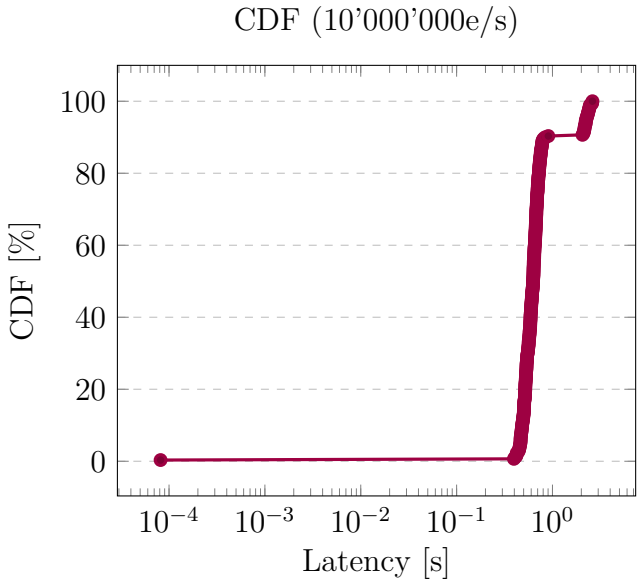

Figure 13: CDF latency plot.

The latency plot reveals a nicely linear behaviour, letting us go beyond ten million events before congesting.

5.3 Remarks

Compared to the queries shown in NEXMark, the Yahoo Streaming Benchmark is exceedingly simple. It also includes some rather odd requirements that were most likely simply specific to Yahoo's internal use-case, rather than born out of consideration for what would make a good benchmark. Most notably the lookup in Redis

would present a significant bottleneck for most modern streaming systems, and the required JSON deserialisation step will lead to the benchmark mostly testing the JSON library's speed, rather than the streaming system's actual performance in processing the data.

Disregarding the Redis look up and the deserialisation, the only remaining operation the benchmark performs is a windowed reduction, as the projection can be mostly disregarded. This means that the benchmark does not add much complexity over a very basic word count test. Thus we believe it is neither representative of typical streaming systems applications, nor extensive enough in testing the system's expressiveness or capabilities.

6 HiBench: A Cross-Platforms Micro-Benchmark Suite for Big Data

HiBench[12] is a benchmarking suite created by Intel in 2012. It proposes a set of microbenchmarks to test Big Data processing systems. It includes implementations of the tests for Spark, Flink, Storm, and Gearpump. For our purposes in testing Strymon, we will focus only on the four tests of the streaming suite:

- **Identity** This test is supposed to measure the minimum latency of the system, by simply immediately outputting the input data.

Figure 14: HiBench's Identity dataflow graph

- **Repartition** This tests the distribution of the workload across workers, but just like Identity does not perform any computation on the data. The repartition should be handled through a round-robin scheduler.

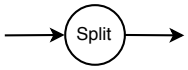

Figure 15: HiBench's Repartition dataflow graph

- **Wordcount** This is a basic word count test that simply focuses on counting the occurrences of individual words, regularly outputting the current tally. It is intended to test the performance of stateful operators.

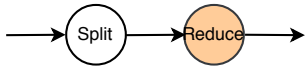

Figure 16: HiBench's Wordcount dataflow graph

- **Fixwindow** This test performs a simple hopping window reduction, with each window being ten seconds long.

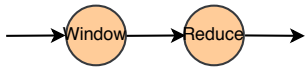

Figure 17: HiBench's Fixwindow dataflow graph

The data used for the benchmark follows a custom CSV-like format, where each input is composed of an integer timestamp and a comma separated list of the following fields:

- `ip` An IPv4 address, presumably for the event origin.
- `session_id` A unique session ID hash.
- `date` Some kind of date in `YYYY-MM-DD` format.
- `?` A float of some kind.
- `user_agent` A browser user-agent string, to identify the user.
- `?` Some three-letter code.
- `?` Some five-letter sub-code.
- `word` A seemingly random word.
- `?` Some integer.

As the benchmark does not publicly state the structure of the workload, and the fields aren't really specifically used for anything in the benchmarks except for the `word`, we can only guess what they are meant to be for.

Since the benchmarks focus on very small tests, they can only really give insight about the performance of the system for a select few individual operations. This might not translate to the performance of the system for complex data flows with many interacting components. Hibench only focuses on the latency component of the system, measuring how long it takes the system to process data at a fixed input rate. It does not consider other important factors of a streaming system such as fault tolerance, scaling, and load bearing.

6.1 Implementation

6.1.1 Data Generation

As the data generation process is not documented explicitly anywhere and the source is rather hard to decode, we opted for a much simpler scheme that should nevertheless follow the overall structure of the data used in HiBench. Our generator produces a fixed-size set of random IPs, by default set to 100. It then generates events at a fixed number of events per second, with the assumption that the timestamps in the data records correspond to seconds. For each event, the `ip` is chosen at random from the set, the `session_id` is a random 54 character long ASCII string, and the `date` is a randomly generated date string in the appropriate format. All of the remaining fields are left the same across all events.

Since, as far as we can tell, only the IP and timestamp are actually used by any of the streaming queries, we do not believe that the lack of proper data generation for the remaining fields severely skews our workloads. This observation is based on the Flink implementation of HiBench.

6.1.2 Data Flows

6.1.2.1 Identity

```
stream.map(|e| (e.time,
↪  SystemTime::now().duration_since(UNIX_EPOCH).unwrap().as_secs()))
```

Listing 29: Implementation for the Identity query.

In this query we simply parse out the timestamp from its string representation and return it alongside the current number of seconds since the UNIX epoch.

6.1.2.2 Repartition

```
let peers = config.get_as_or("threads", 1) as u64;
// Simulate a RoundRobin shuffling
stream.unary_stream(Pipeline, "RoundRobin", move |input, output| {
    let mut counter = 0u64;
    input.for_each(|time, data| {
        output.session(&time).give_iterator(data.drain(..).map(|r| {
            counter = (counter + 1) % peers;
            (counter, r)
        }));
    })})
    // Exchange on worker id (worker ids are in [0,peers)
    .exchange(|&(worker_id, _)| worker_id)
    .map(|(_, record)| record)
```

Listing 30: Implementation for the Repartition query.

This is the most complex implementation of all queries in the HiBench set, since HiBench expects the data exchange to be performed in a round-robin fashion, whereas Timely usually performs a hashing scheme to exchange data between nodes on different workers. There is currently no built-in operator to perform round-robin exchanges, so we have to simulate it with an ad-hoc implementation here. We do this by mapping each input to a tuple of current round-robin count and record. We then use this round-robin count in order to use the usual `exchange` operator. A more efficient implementation would handle the exchange between workers directly.

6.1.2.3 Wordcount

```
stream
    .map(|e| (e.ip(), e.time))
    .rolling_count(|&(ref ip, _)| ip.clone(), |(ip, ts), c| (ip, ts, c))
```

Listing 31: Implementation for the WordCount query.

For a word count, all we really need is the `rolling_count` operator, which performs a continuously updating reduction. In order to achieve a more efficient counting scheme, we exchange each record between workers hashed on the IP. This means that each worker will receive a disjoint set of IPs to count, making the reduction much more efficient.

6.1.2.4 Fixwindow

```
stream
    .map(|e| (e.ip(), e.time))
    .tumbling_window(window_size)
    .reduce_by(|&(ref ip, _)| ip.clone(),
               (0, 0), |(_, t), (m, c)| (min(m, t), c+1))
```

Listing 32: Implementation for the Fixwindow query.

For this query, we merely need to create a tumbling window for ten seconds, and then count the number of events per IP in the window as well as their minimal timestamp, both of which can be achieved with a single `reduce_by`.

6.2 Evaluation

We ran our experiments on an AMD Opteron 6378 2.4KHz 64bit machine with a total of 32 Cores and 504GB RAM. This machine is known to exhibit strange scaling behaviour due to non-uniform memory access patterns. You can see this behaviour in the scaling plot at 16 workers.

Our measurement procedure involved a closed-loop data feed, meaning each epoch was run to completion before a new epoch with a new round of data was started. We only measured data flow execution time, excluding data generation time. Each measurement was gathered using 300 epochs of data.

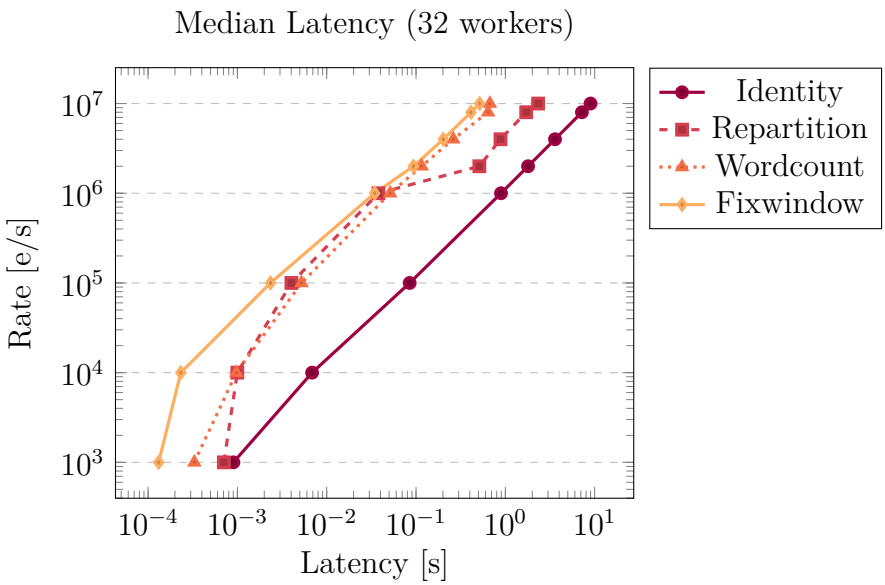

Figure 18: Loglog plot of the Latency vs Rate.

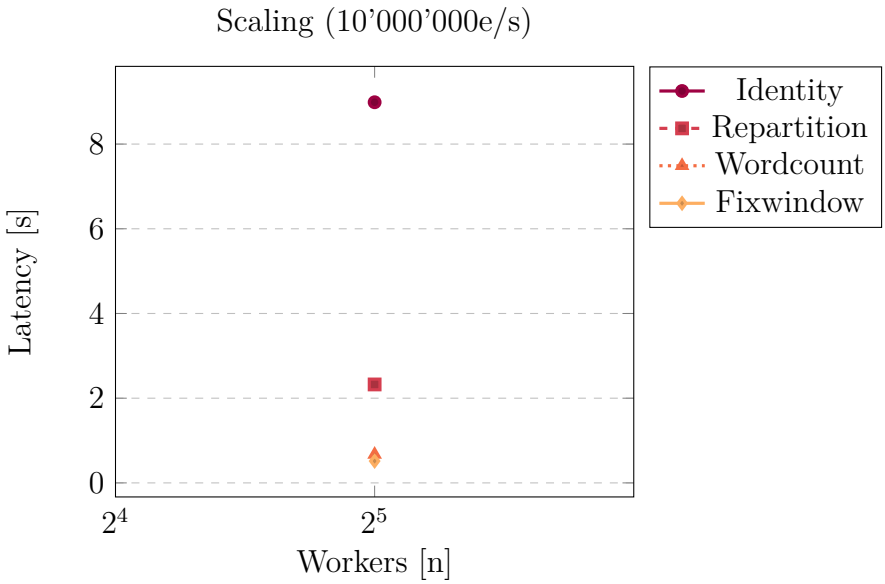

Figure 19: Linlog plot of the scaling behaviour.

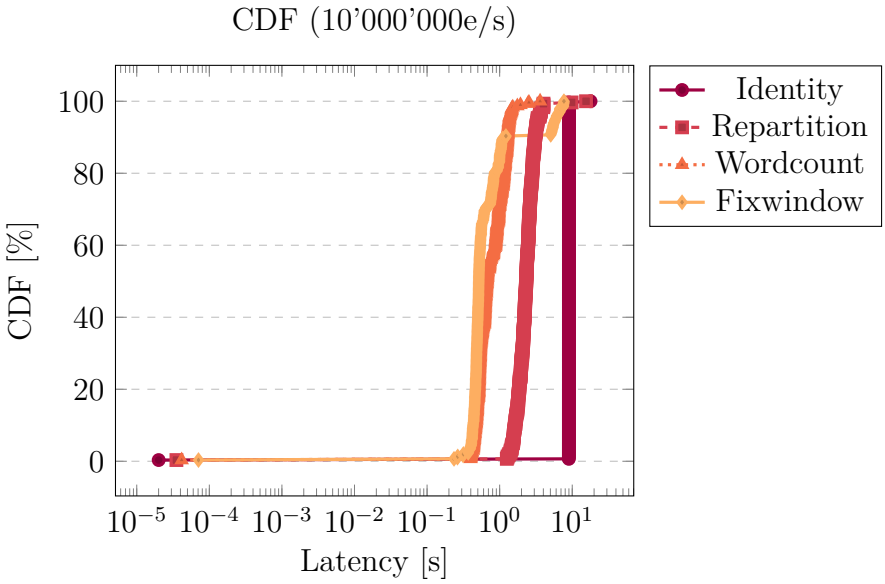

Figure 20: CDF plot.

We surmise that the exceedingly poor performance of the Identity query is down to the requirement of including a current Unix time measurement, which requires a system call that bottlenecks the data flow. The repartition dataflow scales poorly as Timely automatically performs a hash-based redistribution of data for operators that require exchange and we automatically feed data on all workers by default, distributing the data from the get-go.

6.3 Remarks

Surprisingly enough, HiBench gave us a lot of trouble to implement. Not because the queries were complex, but simply because of the lack of proper documentation and maintenance. Beyond the very superficial descriptions of the queries on their homepage, there is nothing about how the workloads are generated, how the queries perform in detail, or what the though process behind the design was. The code base itself is not easy to decipher either, as the information about data generation is distributed over a swath of files, none of which are commented or explained anywhere.

If this didn't already make things bad enough, the benchmark itself does not run on current setups. It requires versions of Hadoop and Kafka that are no longer supported, and does not work under Java 9. We could not get their data generator to work on several systems. We are unsure whether this was due to a misconfiguration somewhere or due to the software being bitrotten, but the authors did not respond to inquiries, and their documentation for the setup of the benchmark is feeble at best.

While we do see some worth in having a very minimal benchmark that focuses on testing the performance of individual operators, we are not convinced that such information could be used to infer meaningful data about a streaming system as a whole, and especially not about its expressiveness and capability to handle larger dataflows.

If a benchmark such as this were formulated again, it is absolutely vital that the authors properly document the data structures used and how they're generated, as well as the exact computation a query should perform. Without this, it is hardly feasible for third-parties to implement the benchmark for their own system and arrive at comparable timing data.

7 NEXMark Benchmark

NEXMark[13] is an evolution of the XMark benchmark. XMark was initially designed for relational databases and defines a small schema for an online auction house. NEXMark builds on this idea and presents a schema of three concrete tables, and a set of queries to run in a streaming sense. NEXMark attempts to provide a benchmark that is both extensive in its use of operators, and close to a real-world application by being grounded in a well-known problem.

The original benchmark proposed by Tucker et al. was adopted and extended by the Apache Foundation for their use in Beam[16], a system intended to provide a general API for a variety of streaming systems. We will follow the Beam implementation, as it is the most widely adopted one, despite having several differences to the benchmark originally outlined in the paper. See subsection 7.3 for an outline of the differences we found. Similar to HiBench and YSB, NEXMark as implemented by Beam does not concern itself with questions of scaling, load bearing, and fault tolerance, focusing solely on the latency aspect.

The benchmark defines the following queries:

0. **Pass-Through** This is similar to HiBench's Identity query and should just output the received data.

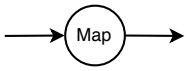

Figure 21: NEXMark's Query 0.

1. **Currency Conversion** Output bids on auctions, but translate the bid price to Euro.

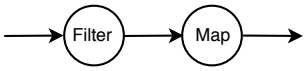

Figure 22: NEXMark's Query 1.

2. **Selection** Filter to auctions with a specific set of IDs.

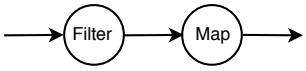

Figure 23: NEXMark's Query 2.

3. **Local Item Suggestion** Output persons that are outputting auctions in particular states.

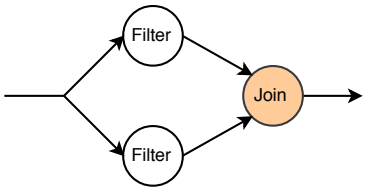

Figure 24: NEXMark's Query 3.

4. **Average Price for a Category** Compute the average auction price in a category for all auctions that haven't expired yet.

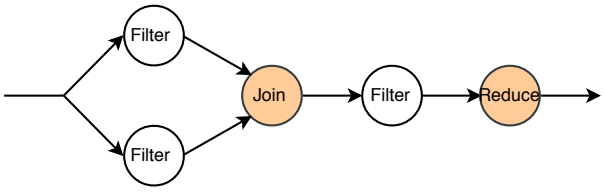

Figure 25: NEXMark's Query 4.

5. **Hot Items** Show the auctions with the most bids over the last hour, updated every minute.

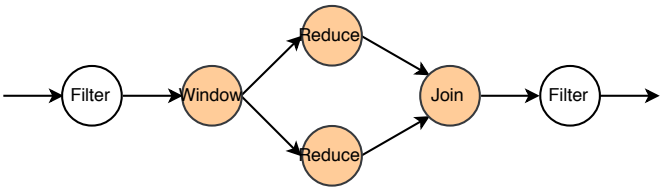

Figure 26: NEXMark's Query 5.

6. **Average Selling Price by Seller** Compute the average selling price for the last ten closed auctions per auctioner.

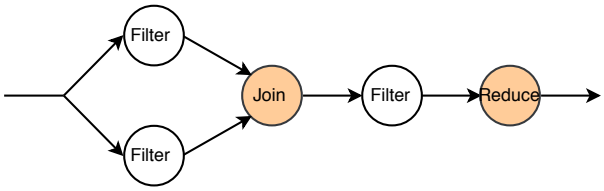

Figure 27: NEXMark's Query 6.

7. **Highest Bid** Output the auction and bid with the highest price in the last minute.

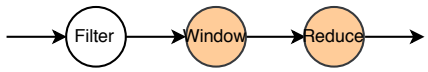

Figure 28: NEXMark's Query 7.

8. **Monitor New Users** Show persons that have opened an auction in the last 12 hours.

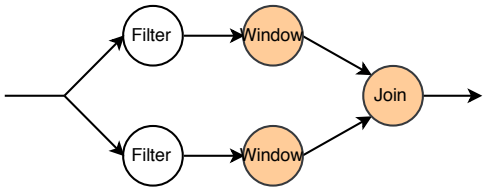

Figure 29: NEXMark's Query 8.

9. **Winning Bids** Compute the winning bid for an auction. This is used in queries 4 and 6.

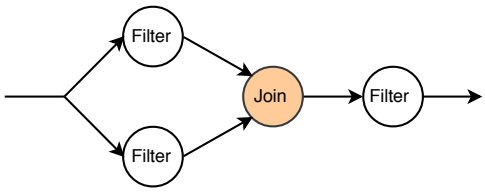

Figure 30: NEXMark's Query 9.

10. **Log to GCS** Output all events to a GCS file, which is supposed to illustrate large side effects.

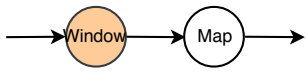

Figure 31: NEXMark's Query 10.

11. **Bids in a Session** Show the number of bids a person has made in their session.

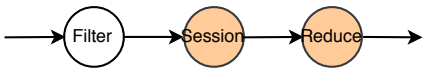

Figure 32: NEXMark's Query 11.

12. **Bids within a Window** Compute the number of bids a user makes within a processing-time constrained window.

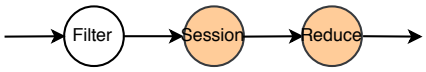

Figure 33: NEXMark's Query 12.

The queries are based on three types of events that can enter the system: `Person`, `Auction`, and `Bid`. Their fields are as follows:

Person

- `id` A person-unique integer ID.
- `name` A string for the person's full name.
- `email_address` The person's email address as a string.
- `credit_card` The credit card number as a 19-letter string.
- `city` One of several US city names as a string.
- `state` One of several US states as a two-letter string.
- `date_time` A millisecond timestamp for the event origin.

Auction

- `id` An auction-unique integer ID.
- `item_name` The name of the item being auctioned.
- `description` A short description of the item.
- `initial_bid` The initial bid price in cents.
- `reserve` The minimum price for the auction to succeed.

- `date_time` A millisecond timestamp for the event origin.
- `expires` A UNIX epoch timestamp for the expiration date of the auction.
- `seller` The ID of the person that created this auction.
- `category` The ID of the category this auction belongs to.

Bid

- `auction` The ID of the auction this bid is for.
- `bidder` The ID of the person that placed this bid.
- `price` The price in cents that the person bid for.
- `date_time` A millisecond timestamp for the event origin.

7.1 Implementation

7.1.1 Data Generation

In order to be able to run the benchmark outside of the Beam framework, we had to replicate their generator. For this we translated the original Java sources of the generator (`sdks/java/nexmark/src/main/java/org/apache/beam/sdk/nexmark/sources/generator`) into Rust. Unfortunately it appears that the Beam generator has a hard constraint on its rate due to a lack of precision, and can thus only output at most 2M events per second. In our tests the system did not congest at these rates, so we had to modify the generation to be more precise, and allow higher rates. This may have changed the generation in subtle ways that we are not aware of. If it is indeed possible to output more than 2M events per second in the original generator, we could not figure out how to make it do so due to the lack of proper documentation.

We have validated our generator against Beam's via manual data output comparison. It should function identical to Beam's setup and our results should thus be accurate even outside of Beam's framework.

7.1.2 Queries

7.1.2.1 Query 0

```
stream.map(|e| e)
```

Listing 33: Implementation for NEXMark's Query 0

7.1.2.2 Query 1

```
stream
    .filter_map(|e| Bid::from(e))
    .map(|b| (b.auction, b.bidder, (b.price*89)/100, b.date_time))
```

Listing 34: Implementation for NEXMark's Query 1

Technically we could have implemented this with a single `filter_map`, but keeping the filtering and the query computation separate is a tad cleaner.

7.1.2.3 Query 2

```
stream
    .filter_map(|e| Bid::from(e))
    .filter(move |b| b.auction % auction_skip == 0)
    .map(|b| (b.auction, b.price))
```

Listing 35: Implementation for NEXMark's Query 2

Similar to paragraph 7.1.2.2, this could have been done with a single `filter_map`, but splitting the work up over multiple operators like this makes the code much easier to read.

7.1.2.4 Query 3

```
let auctions = stream
    .filter_map(|e| Auction::from(e))
    .filter(|a| a.category == 10);

let persons = stream
    .filter_map(|e| Person::from(e))
    .filter(|p| p.state=="OR" || p.state=="ID" || p.state=="CA");

persons.left_join(&auctions, |p| p.id, |a| a.seller,
                  |p, a| (p.name, p.city, p.state, a.id))
```

Listing 36: Implementation for NEXMark's Query 3

This query introduces the first join. Since we need to continuously report on new auctions, we need to retain all persons records indefinitely using the `left_join`.

7.1.2.5 Query 4

```
hot_bids(stream)
    .average_by(|&(ref a, _)| a.category, |(_, p)| p)
```

Listing 37: Implementation for NEXMark's Query 4

The bulk of the work for this query is factored out into the code from paragraph 7.1.2.10. All that remains to do afterwards is to perform an averaging reduce for each category.

It is important to note here that the Beam implementation diverges from the original NEXMark specification by performing a sliding window before the reduction. This change is not documented anywhere. We have decided not to follow this change in our implementation.

7.1.2.6 Query 5

```
let bids = stream
    .filter_map(|e| Bid::from(e))
    .epoch_window(window_size, window_slide)
    .reduce_by(|b| b.auction, 0, |_, c| c+1);

let max = bids.reduce_to(0, |(_, p), c| max(p, c));

max.epoch_join(&bids, |_| 0, |_| 0, |m, (a, c)| (a, c, m))
    .filter(|&(_, c, m)| c == m)
    .map(|(a, c, _)| (a, c))
```

Listing 38: Implementation for NEXMark's Query 5

Query 5 introduces the first use of windowing in NEXMark, and requires a sliding window to boot. The way the filtering for the most bid auction is done may seem a bit odd. Essentially we perform two reductions, once to perform a count of all bids within the epoch, and another time to determine the overall maximum count within the epoch. We then join the two together and filter to only retain the auction that corresponds to the maximum count.

Note that the implementation of this query in Beam differs from our interpretation of the query as described in the original NEXMark paper. As it stands now it emits auctions whose bid count is maximal within the window, whereas the initial query description seems to emit events if and only if there is only one auction mentioned by bids within the window. We consider the initial query to be bogus, since it is practically impossible for an epoch to only contain bids of a single auction.

7.1.2.7 Query 6

```
hot_bids(stream)
    .partition(10, |&(ref a, _)| a.seller)
    .map(|p| (p[0].1, p.iter().map(|p| p.1 as f32).sum::<f32>() / p.len() as
↪   f32))
```

Listing 39: Implementation for NEXMark's Query 6

Like for paragraph 7.1.2.5 most of the work is factored out into the code from paragraph 7.1.2.10. This query is also the only one that makes use of partitioning. The partitioning operator works slightly different from the rest of the windowing-like operators in that it does not unfold its contents into the data stream. Since each epoch might encompass a multitude of partitions, we need to keep the partition's data in vectors. In order to average the data in those records we then have to manually reduce within the map operator.

7.1.2.8 Query 7

```
stream
    .filter_map(|e| Bid::from(e))
    .tumbling_window(window_size)
    .reduce(|_| 0, (0, 0, 0), |b, (a, p, bi)| {
        if p < b.price { (b.auction, b.price, b.bidder) }
        else { (a, p, bi) }
    }, |_, d, _| d)
```

Listing 40: Implementation for NEXMark's Query 7

We make use of the generic reduce operator here in order to quickly determine the maximum priced bid within the window. This could have also been done with an implementation of the `PartialOrd` trait on `Bid` and by using the `maximize_by` operator followed by a map.

7.1.2.9 Query 8

```
let auctions = stream
    .filter_map(|e| Auction::from(e))
    .tumbling_window(window_size);

let persons = stream
    .filter_map(|e| Person::from(e))
    .tumbling_window(window_size);

persons.epoch_join(&auctions, |p| p.id, |a| a.seller,
                   |p, a| (p.id, p.name, a.reserve))
```

Listing 41: Implementation for NEXMark's Query 8

Query 8 is mildly interesting due to the two windows that are joined up. However, due to Timely's epoch based data handling, the synchronisation between the two windows is free.

7.1.2.10 Query 9

```
let bids = stream.filter_map(|e| Bid::from(e));
let auctions = stream.filter_map(|e| Auction::from(e));

let mut auction_map = HashMap::new();
let mut bid_map: HashMap<Id, Vec<Bid>> = HashMap::new();
let auction_ex = Exchange::new(|a: &Auction| a.id as u64);
let bid_ex = Exchange::new(|b: &Bid| b.auction as u64);

auctions.binary_notify(&bids, auction_ex, bid_ex, "HotBids", Vec::new(), move
↪  |input1, input2, output, notificator|{
    input1.for_each(|time, data|{
        data.drain(..).for_each(|a|{
            let future = RootTimestamp::new(a.expires - BASE_TIME);
            let auctions =
↪  auction_map.entry(future).or_insert_with(Vec::new);
            auctions.push(a);
            notificator.notify_at(time.delayed(&future));
        });
    });

    input2.for_each(|_, data|{
        data.drain(..).for_each(|b|{
            bid_map.entry(b.auction).or_insert_with(Vec::new).push(b);
        });
    });

    notificator.for_each(|cap, _, _|{
        if let Some(mut auctions) = auction_map.remove(cap.time()) {
            auctions.drain(..).for_each(|a|{
                if let Some(mut bids) = bid_map.remove(&a.id) {
                    bids.drain(..)
                        .filter(|b| a.reserve <= b.price && b.date_time <
↪  a.expires)
                        .map(|b| b.price)
                        .max()
                        .map(|price| output.session(&cap).give((a, price)));
                }
            });
        }
    });
})
```

Listing 42: Implementation for NEXMark's Query 9

Due to the curious constraints on real-time filtering during a join we opted for implementing a custom operator for this query. The operator proceeds by computing the epoch on which each auction it sees is going to expire. It then remembers the auction for that epoch and schedules a notification to occur at that time. This is useful since we will only be able to emit the joined bid price and auction once the expiry time has been reached and we are sure that we've seen all bids.

On the bid side we simply store each bid on a map associated with the auction it belongs to.

Once we have reached an expiry time, we iterate through each expired auction. We then iterate through all of the auction's bids to filter out invalid ones and compute the maximum bid price. Finally we output the auction associated with the computed price.

7.1.2.11 Query 10

We did not implement Query 10 as we felt it did not reflect a useful case outside of the very specific and particular application of writing to a Google Cloud Storage file.

7.1.2.12 Query 11

```
stream
    .filter_map(|e| Bid::from(e))
    .session(10, |b| (b.bidder, b.date_time / 1000))
    .map(|(b, d)| (b, d.len()))
```

Listing 43: Implementation for NEXMark's Query 11

Since we know for certain that a bid's `date_time` stamp corresponds to an epoch at the second level, we can simply compute the session a bid corresponds to according to that stamp.

7.1.2.13 Query 12

```
let start = Instant::now();
stream
    .filter_map(|e| Bid::from(e))
    .session(10, move |b| {
        let d = Instant::now().duration_since(start);
        (b.bidder, d.as_secs() as usize)
    })
    .map(|(b, d)| (b, d.len()))
```

Listing 44: Implementation for NEXMark's Query 12

We assume that epochs start at `0`, so we can use a timer to measure the elapsed wall clock time since the beginning of the dataflow run and correlate that to a real-time session. Note that for this to work at all, epochs need to be correlated to real-time seconds as well, and a closed-loop experiment with no regards for time constraints will crash.

7.2 Evaluation

We ran our experiments on an AMD Opteron 6378 2.4KHz 64bit machine with a total of 32 Cores and 504GB RAM. This machine is known to exhibit strange scaling behaviour due to non-uniform memory access patterns. You can see this behaviour in the scaling plot at 16 workers.

Our measurement procedure involved a closed-loop data feed, meaning each epoch was run to completion before a new epoch with a new round of data was started. We only measured data flow execution time, excluding data generation time. Each measurement was gathered using 300 epochs of data.

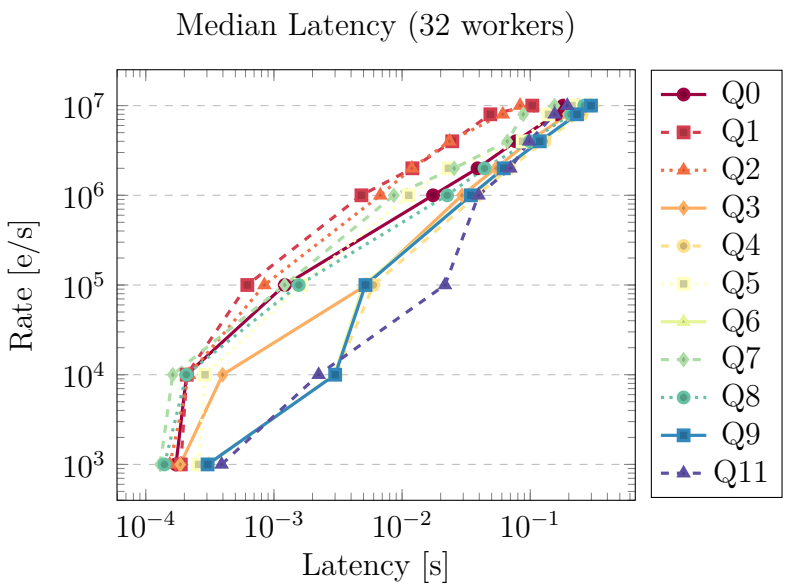

Figure 34: Linlog plot of the Latency vs Rate.

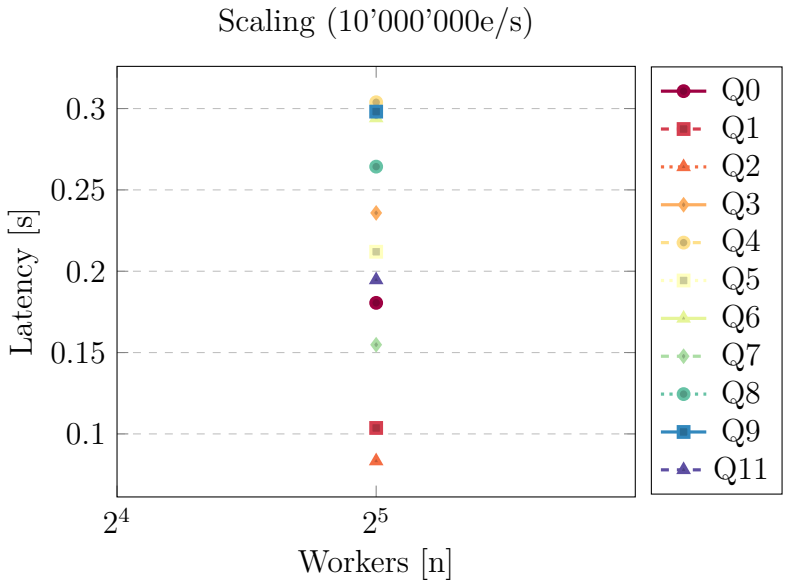

Figure 35: Linlog plot of the scaling behaviour.

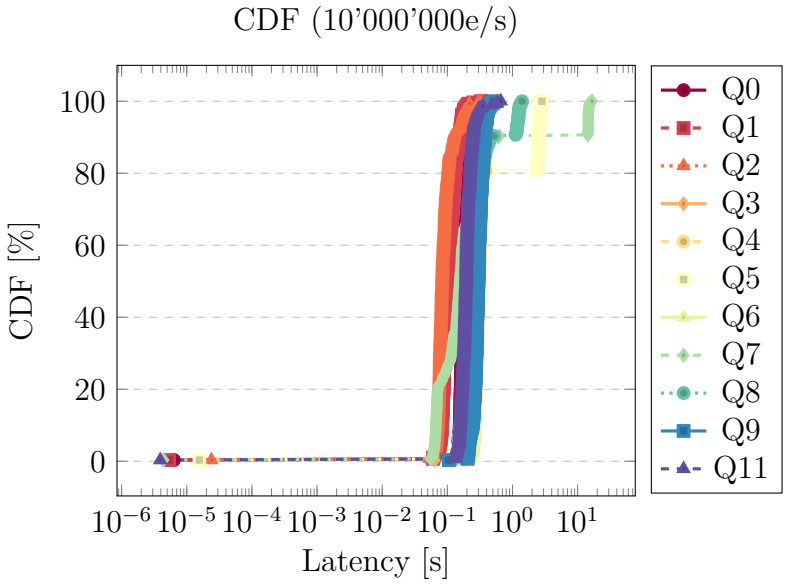

Figure 36: CDF plot.

7.3 Remarks

NEXMark presents the most extensive benchmark of the three we have investigated. It shows a number of different applications that involve a variety of different operators and configurations. Unlike the other two, NEXMark was a research project designed to present a useful benchmark for streaming systems. As such it has a formal specification of the datastructures and queries involved, and presents a reference data generation implementation. These are vital pieces if it should be possible for third-parties to implement the benchmark and compare results.

However, due to the lack of impact of the original paper, the only widespread use of the benchmark is with Apache's Beam system. The Beam implementors made several relatively severe changes to the benchmark. First, they implemented their own generator that has almost nothing in common with the original generator. Second, they added more queries whose precise behaviour and purpose is not formally specified or documented anywhere. Third, they changed the size of the windows to be merely ten seconds, rather than the minutes and hours the original specification sets.

We assume the idea behind Beam's implementation is that, in order to offer comparable benchmarks for systems, you would simply have to write a backend for your system in Beam. This however is not trivially achievable, and also will not actually produce results that will properly reflect your system, as the benchmark will implicitly measure and compare not just your system on its own, but also the backend you wrote for Beam and how well it translates queries.

8 Conclusion

As part of this thesis we have surveyed and evaluated a number of papers describing other, current streaming systems. We specifically looked at the ways in which these systems are used and tested. We found that most tests involve relatively simple data flows and setups, with the exception perhaps being "Big data analytics on high Velocity streams: A case study"[9].

We did not find any consensus on what constitutes a representative, exhaustive, or even just generally sufficient test for a streaming system. The word count example seems to be the most prevalent, but we do not believe that it provides meaningful data for the evaluation of a streaming system when compared to practical problems a system should be capable of solving.

Furthermore we found a disturbing lack of reproducibility of test setups. A large number of papers use data for their experiments that is neither public, nor not described in sufficient detail. We would like to urge authors to either use or provide publicly available datasets, or use well-defined generated data with a publicly available generation algorithm. Without any data available, it is not feasible for peers to validate experiments and make meaningful conclusions about the differences between systems.

For our work we implemented three benchmarks for Timely that have been implemented on a variety of other streaming systems. These benchmarks seem to be among the most widely used to evaluate and contrast performance between systems. To do so we had to implement a number of additional operators in order to concisely express the data flows used in the benchmarks. Thanks to Timely's construction and Rust's expressiveness it is however no big challenge to add even complex oper-

ators to the system that nevertheless perform well.

In our closed-loop experiments evaluating the benchmarks on a 32-core system we found Timely capable of scaling up to tens of millions of events per second before congesting. We believe that this shows great promise for the performance, scalability, and usability of the Timely system as a streaming data processor.

Unfortunately we have also come to the conclusion that most of these benchmarks aren't very useful for the overall evaluation of a streaming systems. They suffer from a lack of specification and documentation (HiBench, NEXMark), include a variety of very questionable operations in their data flows (HiBench, YSB), or do not include data flows that are complex enough to evaluate the system in a meaningful way (HiBench, YSB). Please see the respective remarks sections for a more detailed discussion of the problems we have found.

Based on our experience implementing the aforementioned benchmarks, we believe that the following traits are vital for the definition of a future, useful, and meaningful benchmark:

- Both the input *and output* data schema of each data flow are specified clearly using an abstract specification language. Without this verifying correctness of an implementation is not trivial.
- Workloads are generated according to well-specified, deterministic algorithms with clearly defined parameters and effects. Especially the use of random number generators should be either avoided entirely, or a very specific random number generation algorithm should be specified in detail to be used with the data generation. This is necessary in order to ensure reproducible setups and to make it feasible to verify implementation correctness.
- No specific external system requirements for feeding data into the system, consuming data from the system, or performing any part of the data flow computation. Requiring specific external systems complicates experiment setup, risks implementation bitrot, and introduces outside variables into the evaluation that might significantly bias the performance evaluation, or even bottleneck it.
- Data flows are specified in an abstract modelling language that is well defined. Without a precise definition of the data flows it is impossible to verify whether an implementation is doing the right thing or not. Providing reference implementations for data flows is not acceptable, as the implementation might contain subtle bugs or exhibit other properties of the system that make validation confusing at best, and impossible at worst.
- Each data flow is accompanied by a verified reference output data set using the default parameters. The data set should be available in a common, machine-readable format such as JSON. This allows automated testing of the data generator and the individual data flows to verify correctness.

- Data flows that contain configurable properties include precise descriptions of the effects of the properties, including their valid domains. A lack of clarity on what the effect of a property is makes it hard to estimate what the change of the property is going to evaluate about the system.
- The set of data flows includes a variety of operators in short graphs. This allows the evaluation of individual operators to estimate their costs as basic entities. Including map, filter, reduce, window, and join operators is absolutely vital.
- The set of data flows includes complex graphs that combine a variety of operators. This allows the evaluation of the overall system performance when dealing with deeper graphs and longer computations and shows the system's performance for the application to practical problems.
- The benchmark includes test setups that evaluate not only input to output latency, but also congestion behaviour, scalability, and fault tolerance. Real systems experience load spikes and failures on machines. It is thus essential for a benchmark to include these properties in its evaluation model.

References

[1] Leonardo Neumeyer et al. "S4: Distributed stream computing platform". In: *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*. IEEE. 2010, pp. 170–177.
URL: https://pdfs.semanticscholar.org/53a8/7ccd0ecbad81949c688c2240f2c0c321cdb1.pdf.

[2] Bugra Gedik et al. "SPADE: the system s declarative stream processing engine". In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM. 2008, pp. 1123–1134.
URL: http://cs.ucsb.edu/~ckrintz/papers/gedik_et_al_2008.pdf.

[3] Matei Zaharia et al. "Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters". In: *HotCloud* 12 (2012), pp. 10–10.
URL: https://www.usenix.org/system/files/conference/hotcloud12/hotcloud12-final28.pdf.

[4] Tyler Akidau et al. "MillWheel: fault-tolerant stream processing at internet scale". In: *Proceedings of the VLDB Endowment* 6.11 (2013), pp. 1033–1044.
URL: http://db.cs.berkeley.edu/cs286/papers/millwheel-vldb2013.pdf.

[5] Vincenzo Gulisano et al. "Streamcloud: An elastic and scalable data streaming system". In: *IEEE Transactions on Parallel and Distributed Systems* 23.12 (2012), pp. 2351–2365.
URL: http://oa.upm.es/16848/1/INVE_MEM_2012_137816.pdf.

[6] Raul Castro Fernandez et al. "Integrating scale out and fault tolerance in stream processing using operator state management". In: *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*. ACM. 2013, pp. 725–736.
URL: http://openaccess.city.ac.uk/8175/1/sigmod13-seep.pdf.

[7] Zhengping Qian et al. "Timestream: Reliable stream computation in the cloud". In: *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM. 2013, pp. 1–14.
URL: https://pdfs.semanticscholar.org/9e07/4f3d1c0e6212282818c8fb98cc35fe03f4d0.pdf.

[8] Leonardo Aniello, Roberto Baldoni, and Leonardo Querzoni. "Adaptive online scheduling in Storm". In: *Proceedings of the 7th ACM international conference on Distributed event-based systems*. ACM. 2013, pp. 207–218.
URL: http://midlab.diag.uniroma1.it/articoli/ABQ13storm.pdf.

[9] Thibaud Chardonnens et al. "Big data analytics on high Velocity streams: A case study". In: *Big Data, 2013 IEEE International Conference on*. IEEE. 2013, pp. 784–787.
URL: https://www.researchgate.net/profile/Philippe_Cudre-Mauroux/publication/261281638_Big_data_analytics_on_high_

Velocity_streams_A_case_study/links/5891ae9592851cda2569ec2b/Big-data-analytics-on-high-Velocity-streams-A-case-study.pdf.

[10] Lei Wang et al. "Bigdatabench: A big data benchmark suite from internet services". In: *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*. IEEE. 2014, pp. 488–499.
URL: https://arxiv.org/pdf/1401.1406.

[11] Sanket Chintapalli et al. "Benchmarking streaming computation engines: Storm, Flink and Spark streaming". In: *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*. IEEE. 2016, pp. 1789–1792.
URL: http://ieeexplore.ieee.org/abstract/document/7530084/.

[12] Intel. *HiBench is a big data benchmark suite*.
URL: https://github.com/intel-hadoop/HiBench.

[13] Pete Tucker et al. *NEXMark–A Benchmark for Queries over Data Streams (DRAFT)*. Tech. rep. Technical report, OGI School of Science & Engineering at OHSU, Septembers, 2008.
URL: http://datalab.cs.pdx.edu/niagara/pstream/nexmark.pdf.

[14] F McSherry et al. *Timely dataflow*. 2016.
URL: https://github.com/frankmcsherry/timely-dataflow/.

[15] Derek G Murray et al. "Naiad: a timely dataflow system". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM. 2013, pp. 439–455.
URL: https://dl.acm.org/ft_gateway.cfm?id=2522738&type=pdf.

[16] Apache Foundation. *NEXMark on Apache Beam*.
URL: https://beam.apache.org/documentation/sdks/java/nexmark/.