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

inf | Informatik
Computer Science

Abstract

A real abstract kind of text

Acknowledgements

Cool people

Contents

1 Introduction **2**

2 Related Work **2**
2.1 S4: Distributed stream computing platform[1] 2
2.2 SPADE: the system s declarative stream processing engine[2] 3
2.3 Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters[3] 4
2.4 MillWheel: fault-tolerant stream processing at internet scale[4] 4
2.5 Streamcloud: An elastic and scalable data streaming system[5] 4
2.6 Integrating scale out and fault tolerance in stream processing using operator state management[6] 5
2.7 Timestream: Reliable stream computation in the cloud[7] 6
2.8 Adaptive online scheduling in Storm[8] 7
2.9 Big data analytics on high Velocity streams: A case study[9] 7
2.10 Bigdatabench: A big data benchmark suite from internet services[10] 8
2.11 Comparison 8

3 Timely Dataflow **10**
3.1 Additional Operators 11

4 Yahoo Streaming Benchmark (YSB)[11] **12**
4.1 Implementation 13
4.1.1 Data Generation 13
4.1.2 Query 14
4.2 Evaluation 14
4.3 Remarks 14

5 HiBench: A Cross-Platforms Micro-Benchmark Suite for Big Data[12] **15**
5.1 Implementation 16
5.1.1 Data Generation 16
5.1.2 Queries 17
5.2 Evaluation 18
5.3 Remarks 18

6 NEXMark Benchmark[13] **19**
6.1 Implementation 23
6.1.1 Data Generation 23
6.1.2 Queries 23
6.2 Evaluation 27
6.3 Remarks 27

7 Conclusion **27**

1 Introduction

2 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 nodes are coloured in orange if they are stateful, and covered in red if they perform windowing of some kind and thus retain previous inputs.

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.

2.1 S4: Distributed stream computing platform[1]

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

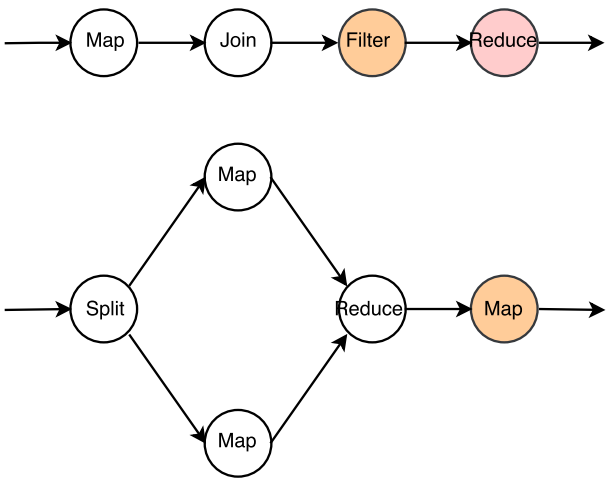


Figure 1: Graphs of the two tests used to evaluate the S4 platform: click-through rate and online parameter optimisation.

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

The OPO test consists of five nodes: the split operator assigns keys to route the events to either one of the map operators. These nodes then perform some computations on the events and emit the results as new events. The reduce node 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.

2.2 SPADE: the system s declarative stream processing engine[2]

In order to evaluate the system, a simple algorithm is run to determine bargains to buy.

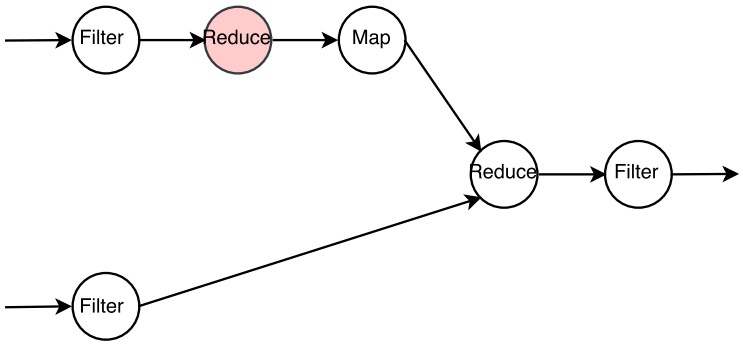


Figure 2: Graph of the example used in the SPADE paper: a bargain index computation.

The data flow is composed of six nodes: a filter node filters out trade information and computes its price. It passes its information on to a moving aggregation node, with a window size of 15, and a slide of 1. The aggregate is passed on to a mapping node that computes the volume weighted average price (VWAP). Another filter node 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.

2.3 Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters[3]

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

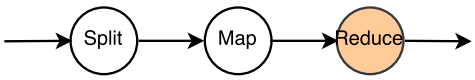


Figure 3: Graph of the Word Count example used to illustrate the discretized streams.

The Word Count test 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 hopping-window aggregation that adds the counters together grouped by word.

2.4 MillWheel: fault-tolerant stream processing at internet scale[4]

The Millwheel paper unfortunately provides barely any information at all about the tests 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 test to measure the latency, and a three-stage test to measure the lag of their fault tolerance system.

2.5 Streamcloud: An elastic and scalable data streaming system[5]

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

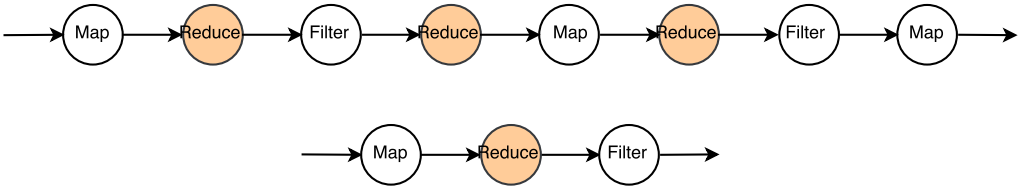


Figure 4: Graph of the query used to evaluate StreamCloud.

Both queries perform a sequence of maps and filters followed by aggregations. The aggregate is based on a window size and slide, which can be configured for each node. However, the configurations used are not provided by the paper.

2.6 Integrating scale out and fault tolerance in stream processing using operator state management[6]

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

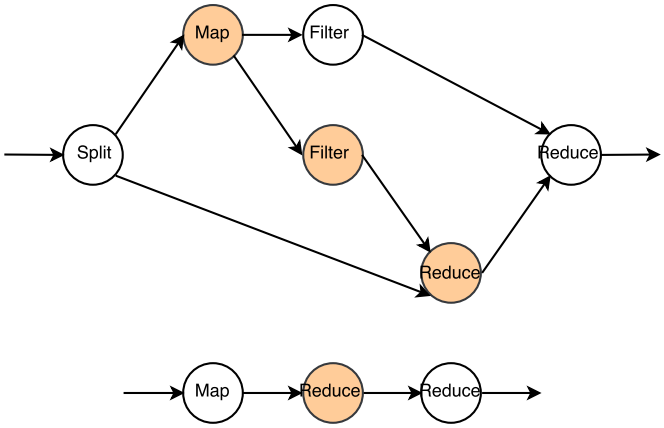


Figure 5: Illustration of the queries for the Linear Road Benchmark and the Top-K tests used to evaluate their system.

The LRB is implemented using six nodes. The first split node routes the tuples depending on their type. The following map node calculates tolls and accidents, the information of which is then forwarded to a node that collects toll information, and a node 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 node.

The Top-K query is implemented using three nodes. The starting map node strips unnecessary information from the tuples. The following node 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.

2.7 Timestream: Reliable stream computation in the cloud[7]

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

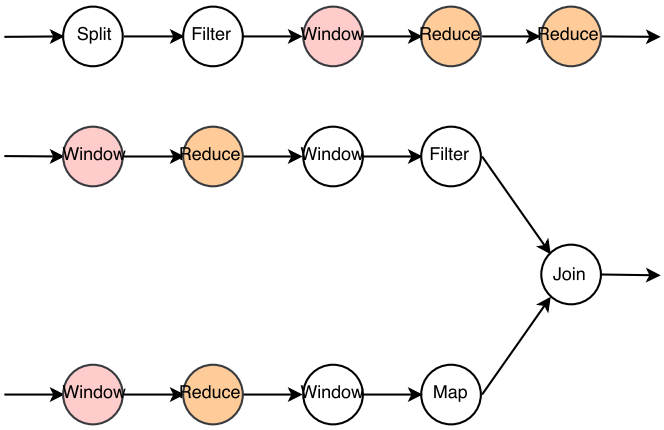


Figure 6: The Distinct Count and Sentiment Analysis queries used to evaluate the Timestream system.

The distinct count is implemented using five nodes. The split node distributes the tuples based on a hash. The following filter removes bot-generated queries, and passes them on to a windowing operator with a window of 30'000 and a slide of 2'000. The windowed events are then reduced into local counts. The local counts are finally aggregated 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 size 2 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 aggregates the word counts for each batch. Using another sliding window of 2 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.

2.8 Adaptive online scheduling in Storm[8]

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

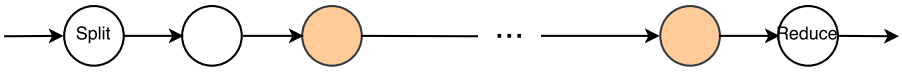


Figure 7: The topology graph used to evaluate the Storm schedulers.

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

2.9 Big data analytics on high Velocity streams: A case study[9]

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

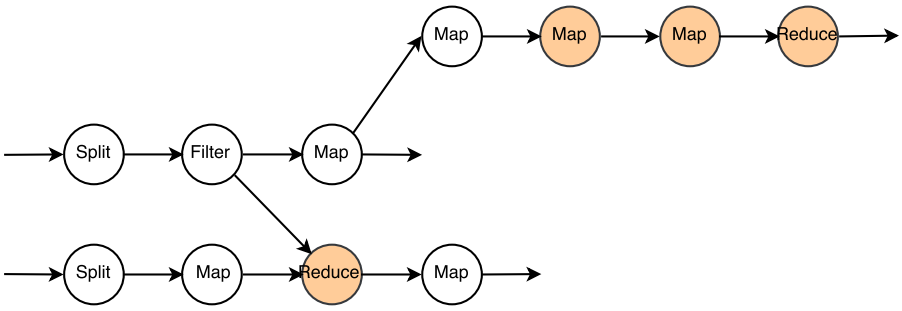


Figure 8: An illustration of the topology used for the Twitter & Bitly link trend analysis.

The data flow for this is the most complicated one presented in the related works we analysed. It uses a total of eleven nodes, excluding edge nodes 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.

2.10 Bigdatabench: A big data benchmark suite from internet services[10]

This paper 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.

2.11 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 test data flows were replicated.

Paper	Goal	Application	Dataflow Properties	Dataflow 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, Reduce, Join
D-Streams[3]	Scalability and recovery test.	None	Chain	Map, Reduce, Window
Millwheel[4]	In-Out Latency.	Ads	Unspecified	Unspecified
StreamCloud[5]	Evaluation of scalability and elasticity.	Telephony	Stateful, Chain	Map, Filter, Reduce, Join
Operator State[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, Window
Adaptive Scheduling[8]	Evaluating performance of scheduling algorithms.	None	Stateful, Chain	None
Analytics on High Velocity Streams[9]	Analysing trends for links on Twitter.	Social Network	Stateful, DAG	Map, Filter, Reduce, 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	**TODO**	**TODO**

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 nodes. Further details not available.	IBM GPFS	Maybe[1]
D-Streams[3]	~20 MB/s/node (200K records/s/node) for WordCount.	Up to 60 Amazon EC2 nodes, 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 nodes, 32 cores, 8GB RAM, 0.5TB disks each, 1Gbit LAN.	Unspecified	No
Operator State[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 nodes, 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 nodes, Intel i7-2600 CPU, 8GB RAM each.	Kafka, Cassandra	Maybe[3]
BigDataBench[10]	Up to 1TB.	15 nodes, 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.

3 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

[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".

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.

3.1 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.

3.1.0.1 FilterMap

This operator performs a filter followed by a map in one go. This is useful when the data stream contains mixed event types and only a single type of event is required. 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.

3.1.0.2 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. The second 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.

3.1.0.3 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` and `average_by` build on top of this to provide more convenient access to reduction. Finally, a separate `reduce_to` does not key data and instead reduces all data within the epoch to a single record.

3.1.0.4 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.

3.1.0.5 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 a window cannot be smaller than a single epoch. When the window is full and the frontier reaches a slide, the window operator sends out a copy of all records within the window.

3.2 The Benchmark Framework

Yadda yadda yodel

4 Yahoo Streaming Benchmark (YSB)[11]

The Yahoo Streaming Benchmark 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.

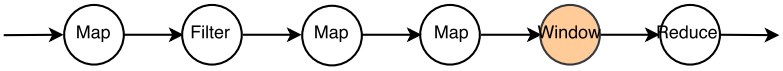


Figure 9: A graph of the dataflow described by YSB.

The dataflow used in the benchmark is illustrated in Figure 9. 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.

4.1 Implementation

4.1.1 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 of our query.

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, reading directly from files gives us more precise data about the performance of Timely itself.

4.1.2 Query

```
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")
         })
    .epoch_window(10, 10)
    .reduce_by(|campaign_id| campaign_id.clone(), 0, |_, count| count+1)
```

Listing 1: 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.

4.2 Evaluation

4.3 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.

5 HiBench: A Cross-Platforms Micro-Benchmark Suite for Big Data[12]

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

The tests are illustrated as data flows in Figure 10. 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.

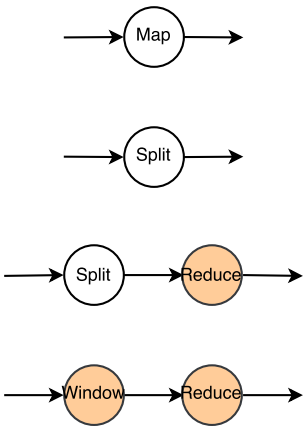


Figure 10: Graphs of the four streaming benchmarks that HiBench specifies.

5.1 Implementation

5.1.1 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.

5.1.2 Queries

5.1.2.1 Identity

```
stream.map(|(ts, _)|
    (u64::from_str(&ts).unwrap(),
     SystemTime::now().duration_since(UNIX_EPOCH).unwrap().as_secs()))
```

Listing 2: 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.

5.1.2.2 Repartition

```
stream.unary_stream(Pipeline, "RoundRobin", move |input, output| {
        let mut counter = 0u64;
        input.for_each(|time, data| {
            for record in data.drain(..) {
                let r = (counter, record);
                counter += 1;
                if counter == peers { counter = 0; }
                output.session(&time).give(r);
            }
        });
    })
    // Exchange on worker id (worker ids are in [0,peers)
    .exchange(|&(worker_id, _)| worker_id)
    .map(|(_, record)| record)
```

Listing 3: 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.

5.1.2.3 Wordcount

```
stream
    .map(|(ts,b)| (get_ip(&b), ts))
    .exchange(|&(ref ip,_)| hasher(&ip))
    .rolling_count(|&(ref ip, _)| ip.clone(), |(ip, ts), c| (ip, ts, c))
```

Listing 4: 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.

5.1.2.4 Fixwindow

```
stream
    .map(|(ts, b)| (get_ip(&b), u64::from_str(&ts).unwrap()))
    .epoch_window(10, 10)
    .reduce_by(|&(ref ip, _)| ip.clone(),
               (0, 0), |(_, t), (m, c)| (min(m, t), c+1))
```

Listing 5: 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`.

5.2 Evaluation

5.3 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.

6 NEXMark Benchmark[13]

NEXMark 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]. 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 SOME-SECTION for an outline of the differences we found.

The benchmark defines the following queries:

0. **Pass-Through** This is similar to HiBench's Identity query and should just output the received data.
1. **Currency Conversion** Output bids on auctions, but translate the bid price to Euro.
2. **Selection** Filter to auctions with a specific set of IDs.
3. **Local Item Suggestion** Output persons that are outputting auctions in particular states.
4. **Average Price for a Category** Compute the average auction price in a category for all auctions that haven't expired yet.
5. **Hot Items** Show the auctions with the most bids over the last hour, updated every minute.
6. **Average Selling Price by Seller** Compute the average selling price for the last ten closed auctions per auctioner.
7. **Highest Bid** Output the auction and bid with the highest price in the last minute.
8. **Monitor New Users** Show persons that have opened an auction in the last 12 hours.
9. **Winning Bids** Compute the winning bid for an auction. This is used in queries 4 and 6.
10. **Log to GCS** Output all events to a GCS file, which is supposed to illustrate large side effects.
11. **Bids in a Session** Show the number of bids a person has made in their session.
12. **Bids within a Window** Compute the number of bids a user makes within a processing-time constrained window.

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

Person

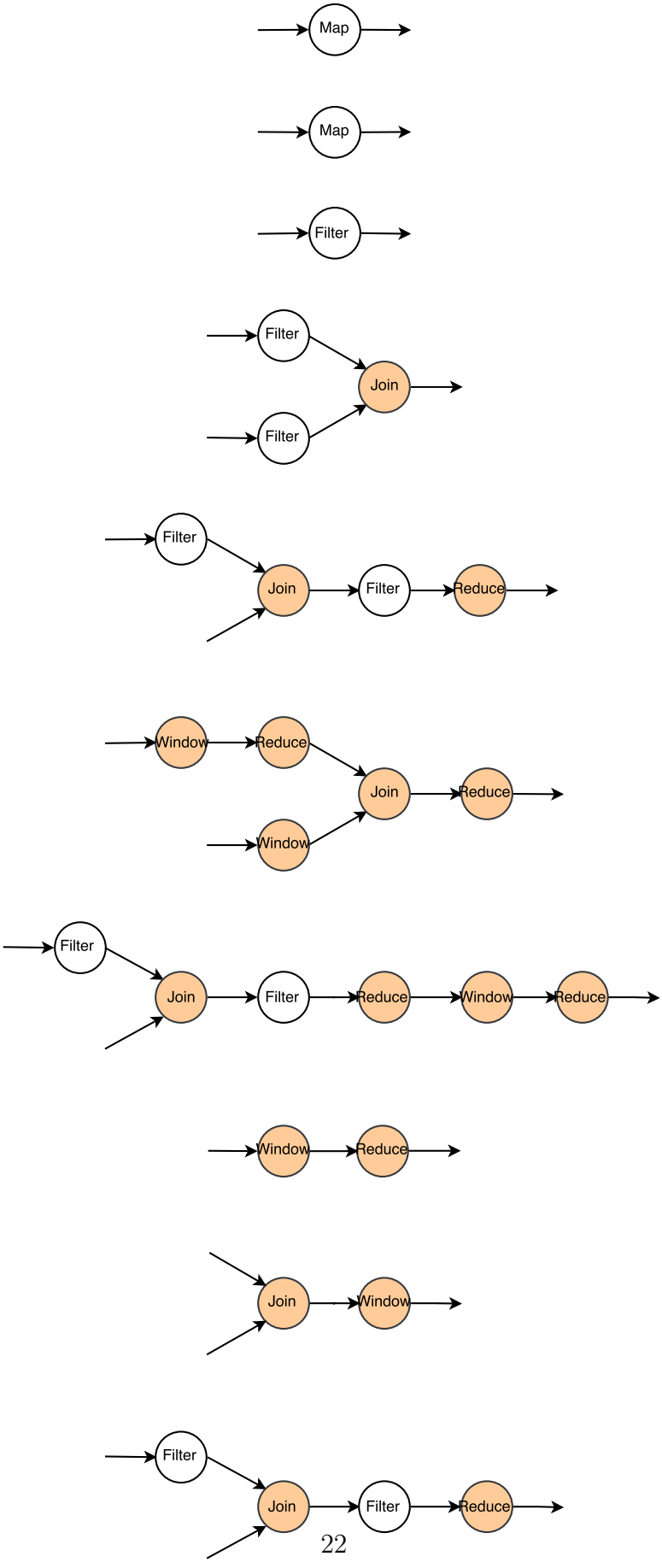
- `id` A person-unique integer ID.
- `name` A string for the person's full name.
- `email_address` The person's email address as a string.
- `credit_card` The credit card number as a 19-letter string.
- `city` One of several US city names as a string.
- `state` One of several US states as a two-letter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


Map
Map
Filter
Filter
Filter
Join
Filter
Filter
Join
Filter
Reduce
Window
Reduce
Join
Reduce
Window
Filter
Join
Filter
Reduce
Window
Reduce
Window
Reduce
Join
Window
Filter
Join
Filter
Reduce

6.1 Implementation

6.1.1 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. We output generated events to aggregate JSON files, where each line is made up of a single event to feed into the system.

6.1.2 Queries

6.1.2.1 Query 0

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

Listing 6: Implementation for NEXMark's Query 0

6.1.2.2 Query 1

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

Listing 7: Implementation for NEXMark's Query 1

6.1.2.3 Query 2

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

Listing 8: Implementation for NEXMark's Query 2

6.1.2.4 Query 3

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

let persons = stream
    .filter_map(|e| e.into())
    .filter(|p: &Person| 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 9: Implementation for NEXMark's Query 3

6.1.2.5 Query 4

```
hot_bids(stream)
    .reduce_by(|&(_, ref a)| a.category, 0,
               |(p, _), c| c + p/NUM_CATEGORIES)
```

Listing 10: Implementation for NEXMark's Query 4

6.1.2.6 Query 5

```
let bids = stream
    .filter_map(|e| e.into())
    .epoch_window(60*60, 60);

let count = bids.reduce_to(0, |_, c| c+1);

bids.reduce_by(|b: &Bid| b.auction, 0, |_, c| c+1)
    .epoch_join(&count, |_| 0, |_| 0, |(a, c), t| (t, a, c))
    .filter(|&(t, _, c)| c >= t)
    .map(|(_, a, _)| a)
```

Listing 11: Implementation for NEXMark's Query 5

6.1.2.7 Query 6

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

Listing 12: Implementation for NEXMark's Query 6

6.1.2.8 Query 7

```
stream
    .filter_map(|e| e.into())
    .epoch_window(60, 60)
    .reduce(|_| 0, (0, 0, 0), |b: Bid, (a, p, bi)| {
        if p < b.price { (b.auction, b.price, b.bidder) }
        else { (a, p, bi) }
    }, |_, d, _| d)
```

Listing 13: Implementation for NEXMark's Query 7

6.1.2.9 Query 8

```
let auctions = stream
    .filter_map(|e| e.into())
    .epoch_window(60*60, 60*60);

let persons = stream
    .filter_map(|e| e.into())
    .epoch_window(60*60, 60*60);

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

Listing 14: Implementation for NEXMark's Query 8

6.1.2.10 Query 9

```
let bids = stream.filter_map(|e|{let b: Option<Bid>=e.into(); b});
let auctions = stream.filter_map(|e|{let a: Option<Auction>=e.into(); a});

let mut auction_map = HashMap::new();
let mut bid_prices: HashMap<Id, usize> = HashMap::new();

auctions.binary_notify(&bids, Pipeline, Pipeline, "HotBids", Vec::new(),
↪  move |input1, input2, output, notificator|{
    input1.for_each(|time, data|{
        data.drain(..).for_each(|a: Auction|{
            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|{
            // FIXME: Check if (B.date_time < A.expires)
            if let Some(other) = bid_prices.remove(&b.auction) {
                bid_prices.insert(b.auction, max(other, b.price));
            } else {
                bid_prices.insert(b.auction, b.price);
            }
        });
    });

    notificator.for_each(|cap, _, _|{
        if let Some(mut auctions) = auction_map.remove(cap.time()) {
            auctions.drain(..).for_each(|a|{
                if let Some(price) = bid_prices.remove(&a.id) {
                    output.session(&cap).give((price, a));
                }
            });
        }
    });
})
```

Listing 15: Implementation for NEXMark's Query 9

6.1.2.11 Query 10

```
use serde_json;
```

Listing 16: Implementation for NEXMark's Query 10

6.1.2.12 Query 11

```
use serde_json;
```

Listing 17: Implementation for NEXMark's Query 11

6.1.2.13 Query 12

```
use serde_json;
```

Listing 18: Implementation for NEXMark's Query 12

6.2 Evaluation

6.3 Remarks

7 Conclusion

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/.