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 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).

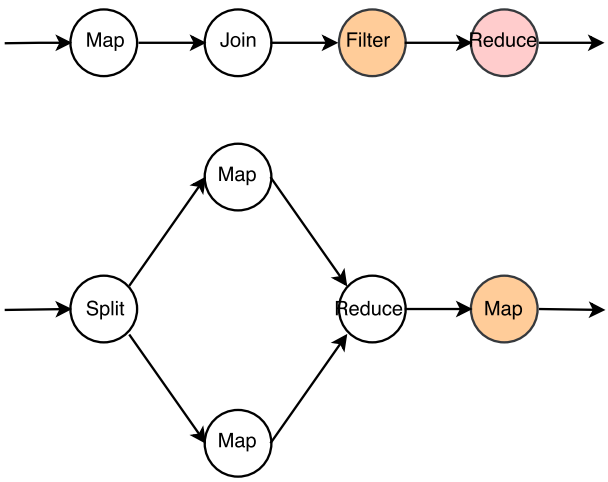

Figure 1: Graphs of the two tests used to evaluate the S4 platform: click-through rate and online parameter optimisation.

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.

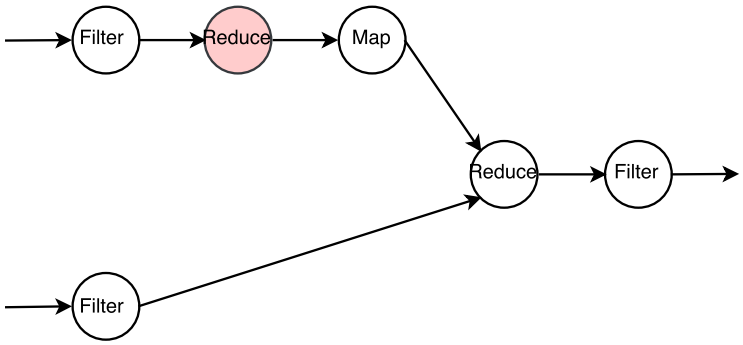

Figure 2: Graph of the example used in the SPADE paper: a bargain index computation.

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.

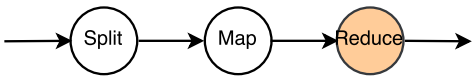

Figure 3: Graph of the Word Count example used to illustrate the discretized streams.

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.

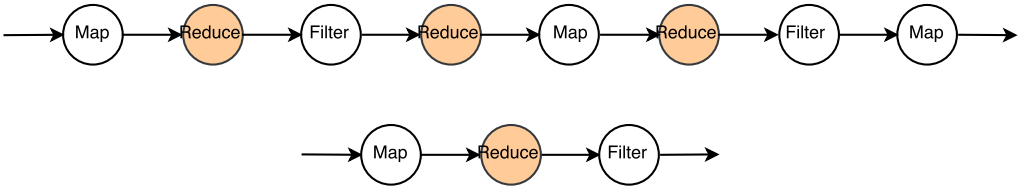

Figure 4: Graph of the query used to evaluate StreamCloud.

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.

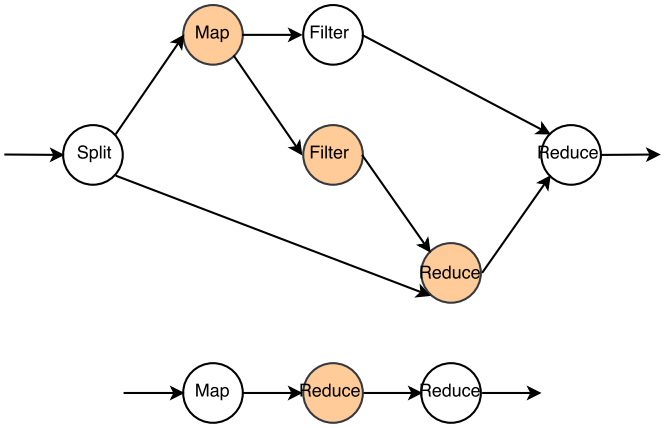

Figure 5: Illustration of the queries for the Linear Road Benchmark and the Top-K tests used to evaluate their system.

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.

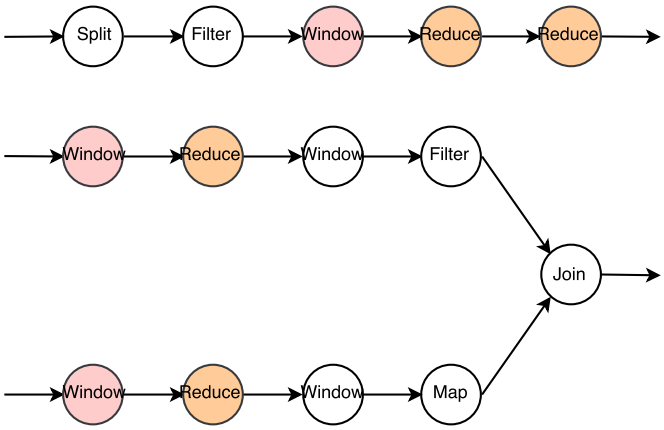

Figure 6: The Distinct Count and Sentiment Analysis queries used to evaluate the Timestream system.

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.

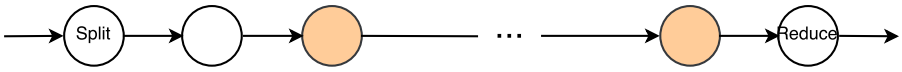

Figure 7: The topology graph used to evaluate the Storm schedulers.

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.

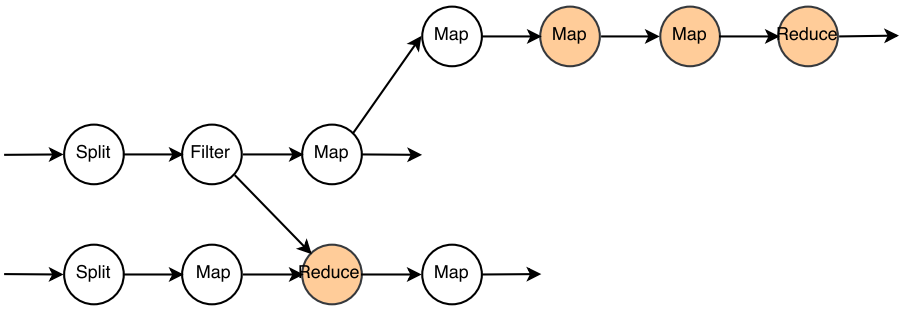

Figure 8: An illustration of the topology used for the Twitter & Bitly link trend analysis.

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

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.

[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".

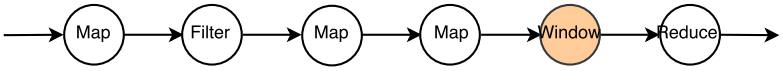

Figure 9: A graph of the dataflow described by YSB.

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

```
let stream = stream
    // Filter to view event_type events.
    .filter(|x: &Event| x.event_type == "view")
    // Transform/Project to ad_id and event_time.
    .map(|x| (x.ad_id, x.event_time))
    // Join the ad_id into the campaign_id through a table lookup.
    .map(move |(ad_id, _)|
         match table.get(&ad_id){
             Some(id) => id.clone(),
             None => String::from("UNKNOWN AD")
         })
    // Aggregate to 10s windows based on 1s epochs.
    .epoch_window(10, 10)
    // Count each campaign in the window and return as tuples of id +
↪   count.
    .reduce_by(|campaign_id| campaign_id.clone(), 0, |_, count| count+1);
```

Listing 1: Dataflow implementation of the YSB benchmark.

4.2 Evaluation

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.

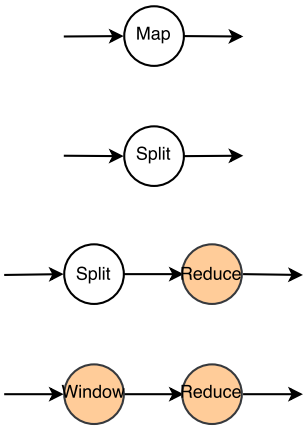

Figure 10: Graphs of the four streaming benchmarks that HiBench specifies.

5.1 Implementation

```
stream.map(|(ts,_):(String,_)|
    (u64::from_str(&ts).expect("Identity: Cannot parse event timestamp."),
↪  SystemTime::now().duration_since(UNIX_EPOCH).expect("Time went
↪  backwards").as_secs()))
    .capture_into(producer);
```

Listing 2: Implementation for the Identity query.

```
// Simulate a RoundRobin shuffling
let stream = 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.

```
let stream = stream
    .map(|(ts,b)| (get_ip(&b),ts))
    .exchange(|&(ref ip,_)| hasher(&ip))
    .rolling_count(|&(ref ip,ref ts):&(String,String)|
↪  (ip.clone(),ts.clone()));
```

Listing 4: Implementation for the WordCount query.

```
let stream = stream
    .map(|(ts,b):Self::D|
↪   (get_ip(&b),u64::from_str(&ts).expect("FixWindow: Cannot parse event
↪   timestamp.")))
    // TODO (john): Check if timestamps in the input stream correspond to
↪   seconds
    // A tumbling window of 10 epochs
    .epoch_window(10, 10)
    // Group by ip and report the minimum observed timestamp and the total
↪   number of records per group
    .aggregate::<_,(u64,u32),_,_,_>(
       |_ip, ts, agg|
        {
            agg.0 = cmp::min(agg.0,ts);
            agg.1 += 1;
        },
       |ip, agg| (ip, agg.0,agg.1),
       |ip| hasher(ip)
    );
```

Listing 5: Implementation for the Fixwindow query.

5.2 Evaluation

6 NEXMark Benchmark[13]

6.1 Implementation

6.2 Evaluation

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.