Of Streams and Storms

A Direct Comparison of IBM InfoSphere Streams and Apache Storm in a Real World Use Case – Email Processing

Zubair Nabi and Eric Bouillet
IBM Research Dublin

Andrew Bainbridge and Chris Thomas
IBM Software Group Europe

April 2014

Table of Contents

Executive Summary

The past few years have witnessed an unparalleled surge in both structured and unstructured data being generated by heterogeneous sources. These sources vary from scientific computations and sensor network deployments to high frequency financial markets and Web 2.0 applications. This has in tandem engendered an entire ecosystem of high-level computation frameworks targeting batch, streaming, iterative, and incremental applications, which abstract away the details of distributed computation underneath simple APIs. In a similar vein, the need to store datasets pre- and post-analysis has led to innovation in traditional DBMS, NoSQL stores, and distributed file systems. On the infrastructure side, due to economies of scale, the computation and storage model is supported by commodity off-the-shelf hardware.

A key goal of organizations which crunch these datasets, is to get timely results with sub-second latency. This real-time computation requirement is naturally fulfilled by stream processing systems, which enable analysis on *data in motion*. IBM InfoSphere Streams is an award-winning product in this domain, enabling line-rate processing of real-time data streams for an extremely wide range of analytics because it both provides out of the box analytic capabilities and allows for easy extension with Java and C++ logic. The underlying engine is supported by the Stream Processing Language (SPL), which allows practitioners to define complex analytic applications using very simple constructs. This combined with a custom Integrated Development Environment (IDE) facilitates a rich ecosystem for stream processing where the practitioner primarily focuses on the desired analytics while the platform does all the heavy lifting.

In this paper, we compare the performance of IBM InfoSphere Streams against Apache Storm [1], a leading open source alternative, to augment existing literature [2]. To this end, we implemented a real-world stream processing application, which enables email classification for online spam detection [3] on both platforms. Our goal was to analyze both the quantitative differences in performance as well as the qualitative differences in application writing and framework tuning. Similar to other studies [4, 5], we employed CPU time and throughput as primary metrics to compare the efficacy of both systems. Overall, our results show that for the application benchmark documented in this paper, Streams outperforms Storm by **2.6 to 12.3** times in terms of throughput while simultaneously consuming **5.5** to **14.2** times less CPU time.

The measurements we made and the particular system environments used are described in this paper. Notably, the outcomes from this study show that the CPU time consumed by Streams to process the entire benchmark dataset is less than what is consumed by Storm to process approximately one quarter of the same dataset. Furthermore, even with application logic removed (pushing raw data through the pipeline), the outcomes from this study show that Streams significantly outperforms Storm, for the benchmark and associated data.

It should be emphasized that significant efforts were made to tune and optimize the Storm version of the benchmark and to overcome difficulties that were encountered when running larger workloads with Storm (see Section 3.2.2). We estimate that more than a month was spent trying to improve Storm performance, well over twice the time spent optimizing the Streams version.

1 Introduction

Email has become a primary and pervasive source of both personal and business communication on the Internet. At the same time, it is the de-facto medium for spam dissemination; so much so that more than 90% of the total global email volume is spam [6]. The annual cost of this unwanted and at times malicious content to just organizations in the United States is estimated to be in excess of $20 billion [7]. To remedy the situation, organizations employ sophisticated spam detection solutions to stem the spam tide. These solutions rely on statistical features derived from email content and metadata to classify emails. To enable organizations and email service providers to stay one step ahead of the spammers in their "cat and mouse" game [6], it is imperative for the former to perform detection on the fly. For this purpose, we designed a real-time statistical features calculation pipeline for streaming email content, which will act as a pre-processing phase for a larger spam detection system.

First, we outline some design goals. The application should mimic the storage and retrieval behavior of real-world deployments, such as employing compression and serialization. Moreover, the pipeline should perform filtering, modification, and metric calculations that can readily be used by a practical spam detection solution [3]. In addition, the application should take advantage of natural stream processing constructs such as cross-stream processing, running window operations, and pipelining. Finally, it should make use of a practical dataset to capture real-world data characteristics.

Behavioral-based spam classification relies on statistics calculated over an email corpus to differentiate between normal and spam emails. These statistical features are calculated on both a per email basis as well as rolling window fashion, i.e. the last 20 emails sent by a user. For instance, certain worms use fake MIME types to pass under the radar of email scanners. As a result, the MIME type of an email is a useful statistical feature. Similarly, counting the number of words and characters in the email body helps the classifier build a profile of a user's writing habits. Furthermore, most users do not send consecutive emails with attachments. For that reason, keeping track of the number of attachments sent by a user in a running window is another statistical feature. Finally, a number of techniques make use of clean datasets, i.e. those that do not contain any spam, and then manually inject spam to simulate worm activity [3]. Section 3 maps some of these requirements to the design of our benchmark. It is noteworthy that due to the wide variance in the techniques used by different algorithms, we do not focus on one particular instance but rather take a broad sweep of pre-processing requirements. Therefore, instead of targeting a particular use-case, our benchmark is an illustrative example of how stream-processing systems can aid in spam detection.

The rest of this document is organized as follows: We give an overview of the stream processing frameworks used for our benchmarks in Section 2. We then present the implementation details of our benchmarks in Section 3. In Section 4 we discuss the results of the performance measurements. Our conclusions are presented in Section 5.

2 Framework Features Overview

In this section we provide an overview of IBM InfoSphere Streams and Storm and where applicable, also highlight useful features of both frameworks.

2.1 IBM InfoSphere Streams

IBM InfoSphere Streams (from hereon called Streams) is IBM’s flagship, high-performance stream processing system that integrates 10 years of research and engineering. Its unique design and implementation enables it to ingest streaming and real-time data at a large scale and process it at line rate while exposing a *lingua franca* for stream processing. As a result, it has found applications in a diverse array of domains including transportation [8], speech analysis [9], DNA sequencing [10], radio astronomy [11], weather forecasting [12], and telecommunications [13]. The Stream Processing Language (SPL) atop Streams enables practitioners to define complex applications as a series of discrete transformations (in the form of *Operators*) over potentially multiple data streams – which flow from “Sources” to “Sinks” -- while abstracting away the intricacies of distributed execution. With a built-in library and assorted toolkit of common stream processing constructs and domain-specific utilities, an application can be up and running in a matter of minutes. Along with SPL itself, users have the option of defining their operators in C++ and Java, or other languages that can be wrapped into these two programming languages, such as Fortran and scripting languages. In addition, Streams Studio is an all-encompassing development and management environment that streamlines development to drag and drop placement of existing components while exposing a central portal for cluster management and monitoring.

Architecturally, Streams consists of management services that take care of different aspects of distributed state management, such as authentication, scheduling, and synchronization. Streams jobs are executed within *Processing Elements*, which can further be fused together to optimize for low latency. Furthermore, the standard TCP based queuing mechanism can be replaced with a built-in high performance Low Latency Messaging (LLM) to further optimize applications in certain environments. The reader is directed to [14] for an exhaustive list of features and development guidelines.

2.2 Storm

Storm is an open-source stream-processing framework originally developed by Backtype and subsequently at Twitter. While the underlying framework is in Clojure, a Storm application -- called a *Topology* -- can be written in any programming language, with Java as the predominant language of choice. Users are free to stitch together a directed graph of execution, with *Spouts* (data sources) and *Bolts* (operators). Architecturally, it consists of a central job and node management entity dubbed the *Nimbus* node, and a set of per-node managers called *Supervisors*. The Nimbus node is in charge of work distribution, job orchestration, communication, fault-tolerance, and state management (for which it relies on ZooKeeper [15]). The parallelism of a Topology can be controlled at 3 different levels: number of *workers* (cluster wide processes), *executors* (number of threads per worker), and *tasks* (number of bolts/spouts executed per thread). Intra-worker communication in Storm is enabled by LMAX Disruptor [16] while ZeroMQ[1] [17] is employed for inter-worker communication. Moreover, tuple distribution across tasks is decided by *groupings*; with *shuffle grouping*, which does random distribution, being the default option.

[1] Since the time of our initial work, which employed Storm 0.8, Storm has been modified to optionally use Netty as a transport in its current version: 0.9 – we show similar results in Section 4.5.1 for Storm 0.9.

3 Benchmark Design

Based on the design goals enumerated in Section 1, we now present the design of our test benchmark. For the dataset, we employ the canonical Enron email dataset [18], which was released publicly as part of the Enron Corporation investigation. The dataset (316 MB compressed, 1.1 GB uncompressed) we used consists of nearly half a million emails belonging to 158 individuals. It contains a good range of sizes of emails, ranging from very small to body sizes of over 1.5 MB.

The data was subjected to an offline cleaning and staging phase in which all emails were serialized, compressed, and stored within a single file. Apache Avro [19] was employed as the serialization framework due to its compact size and fast performance [20] making it an excellent candidate for use in streaming applications. Figure 1 shows the Avro schema that was used for defining an Email object. For compression we used the ubiquitous GZIP standard. It is also important to highlight that separate input files were generated for Storm and Streams in formats *native* to the respective language runtimes. Specifically, in case of Storm, emails were written using a standard Java `GZIPOutputStream` wrapped in an `ObjectOutputStream`. For Streams, a `FileSink` operator with format `bin` and compression `gzip` was employed to this end. Along with simple encoding/decoding for both runtimes, this also enabled each email to be serialized in a format amenable to straightforward conversion to a tuple specific to each platform. It is noteworthy that while seemingly these formats are different, under the hood both follow a similar scheme and the size of the resulting files was almost identical. For instance, the bin format in SPL and the ObjectOutputStream in Java both use a similar mechanism for encoding strings which consists of first encoding the length of the string followed by the string data using UTF. A similar scheme is also used by SPL and ObjectOutputStream to encode byte arrays: the object type that we used for both datasets. Therefore, the use of these formats maintains a level playing field while improving programmer productivity by allowing the use of existing encoding constructs.

```
{ "namespace": "storm.enron.avro",
  "type": "record",
  "name": "Email",
  "fields": [
    { "name": "ID",        "type": "string" },
    { "name": "From",      "type": "string" },
    { "name": "Date",      "type": "string" },
    { "name": "Subject",   "type": ["string", "null"] },
    { "name": "ToList",    "type": "string" },
    { "name": "CcList",    "type": ["string", "null"] },
    { "name": "BccList",   "type": ["string", "null"] },
    { "name": "Body",      "type": ["string", "null"] },
    { "name": "CharCount", "type": ["int", "null"] },
    { "name": "WordCount", "type": ["int", "null"] },
    { "name": "ParaCount", "type": ["int", "null"] }
  ]
}
```

Figure 1: Avro Schema

The processing pipeline for the benchmark email classification system (shown in Figure 2) is divided into 7 stages based on distinct functionality and to optimize the tradeoff between parallelism and data transfer overhead. As mentioned earlier, this pipeline calculates statistical features on emails for classification. For instance, the number of words and characters in the email body constitute a representative profile of content, which helps in disambiguating normal email from spam. The results of the email classification system could be used by spam detection algorithms, such as Naïve Bayes [21]. We now describe each stage in detail.

File Read and Decompress:
Read an email tuple and decompress it into a binary object. It is noteworthy that while in a real-world deployment these emails will be streamed from a message queue, in our experiments disk I/O was not a bottleneck, i.e. input tuples were read at a much faster rate than the processing rate of the subsequent pipeline. Therefore, for simplicity we avoided the use of a message queue and directly read from file.

Deserialize:
Use an Avro library to deserialize the binary object into an Email tuple.

Filter:

- Drop emails that did not originate within Enron, i.e. email addresses that do not end in @enron.com. We assume that all emails originating within the organization are spam free. Confining the dataset to spam free emails allows Bayesian-based spam filtering to use it as ground truth for classification.
- Complementary to dropping emails that did not originate within Enron, remove all email addresses that do not end with @enron.com from the To, CC, and BCC fields.
- Remove rogue formatting such as dangling newline characters and MIME quoted-printable characters [22] from the email body to restrict the character set to simple ASCII.

Modify:

- Obfuscate the names of three individuals in the email body by replacement with *Person1*, *Person2*, and *Person3*. This masking is useful in a real-world environment to protect the privacy of individuals without sacrificing the efficacy of the data or reducing its fidelity.
- Work out the most frequent word in the email body and prepend it to the subject field as an illustrative task for "Per-email Continuous Features" [3].

Metrics:

- Count the total number of emails.
- Count the number of characters, words, and paragraphs in the email body processed in a running window. This is an example of "Features Calculated Over a Sending Window" [3].
- Count the number of characters, words, and paragraphs and total number of emails across multiple streams.

Serialize:
Use an Avro library to serialize each Email tuple into a binary representation.

Compress and Write:
GZIP compress the binary email tuple and write to `/dev/null`[2].

[2] A special file in Unix systems, which discards all data written to it. Normally the results would be pipelined to a downstream application such as the spam detection algorithms or a persistent store.

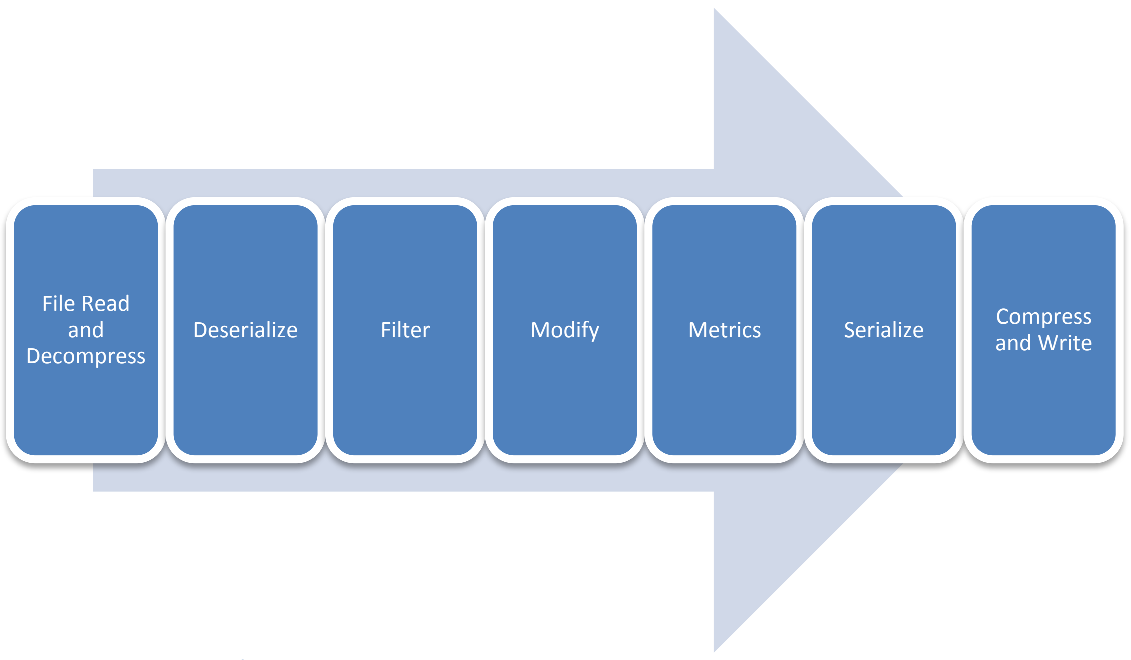


Figure 2: Processing Pipeline

3.1 Streams Implementation

The Streams version of the benchmark application had four main design points:

- Use standard SPL constructs (with just one exception; see below).
- Implement the main components (filter, modify, etc.) of the application logic as distinct Streams operators.
- Simplify the implementation of parallelism by duplicating the entire processing stream (not just individual operators).
- Facilitate easy monitoring and performance analysis by configuring a separate Streams processing element (PE) for each operator.

The exception to the "standard SPL constructs only" design point was the use of two Streams primitive operators (i.e. operators implemented in C++ or Java rather than SPL) to interface to the C++ routines used to process Avro-encoded data. The model for implementing such operators gives a straightforward way to achieve integration with external libraries.

3.1.1 Streams Application Structure

The benchmark scenario is implemented in Streams as the following sequence of operators:

A Streams *FileSource* operator to read and decompress the file containing the emails, emitting an SPL `blob` binary data object.

The *AvroDecode* primitive operator to deserialize the Avro-encoded emails, emitting an SPL tuple of user-defined type `emailTuple` with `rstring`s (SPL string representation) representing each of the eight fields of the email (described above).

emailFilter, a custom operator (written in SPL) that drops email not sent from enron.com addresses, strips out any To, CC, or BCC addresses outside enron.com (by tokenizing the addresses in each field), and removes undesired characters using regex matching.

emailModify, a custom operator that (a) implements the name obfuscation using regex matching; and (b) identifies the most frequently used word in the email body (using a hash table to record each distinct word), appending it to the email's subject field.

The *emailModify* operator emits two separate processing streams: the modified email tuple and a copy of the email body represented as a list of integer character codes.[3] The modified email tuples are processed by two final operators:

The *AvroEncode* primitive operator, which Avro-serializes the email tuples and emits an SPL `blob` binary data object.

A Streams *FileSink* operator which compresses the binary data and writes it to a file (`/dev/null`, to avoid unwanted I/O).

The email body tuples are processed by a single operator:

emailCount, a custom operator that counts the number of characters, words and paragraphs in each email body, creates and emits a tuple of user-defined type `emailMetrics` with `integer`s representing each of the three values.

The above seven operators implement the main processing stream. An eighth sink operator, *metrics,* aggregates the counts for all emails. When multiple streams of data are being processed in parallel, there is an instance of the sequence of seven operators for each stream, but just one instance of the *metrics* operator (in order to maintain a global count of metrics across all streams).

Configuration of multiple processing streams is achieved using a feature of the SPL compiler, namely the ability to use the Perl language to pre-process the SPL source code. In this case, the Perl pre-processor is used to create multiple instances of the seven operators that comprise the main processing stream (modeled on the *SplitParellizer* example in the Streams documentation). The number of processing streams is simply specified as a pre-processor parameter[4]. When more than one stream is configured, a Streams *Filter* operator is used as an efficient mechanism to join the *emailMetrics* tuple flows from the multiple instances of the *emailCount* custom operator.

3.1.2 Streams Benchmark Development and Optimization

The Streams implementation of the benchmark scenario was prototyped very rapidly; the first version (excluding the Avro primitive operators) took less than four days' effort. It is

---

[3] Because of how email body handling was implemented (see Section 3.1.1), this duplication of the email body data was felt necessary to give a more straightforward application design, at the cost of additional processing overhead (duplicated serialization, transmission and deserialization of the email body).

[4] The newly released User Defined Parallelism (UDP) feature in Streams makes parallelizing an entire region of the graph even easier with a parameterized in-line annotation – UDP was not yet available at the time of our analysis.

also worth noting that this work was undertaken by a person with somewhat rusty programming skills and no prior SPL experience.

Once obvious inefficiencies were eliminated from this initial version, relatively little tuning was needed to achieve the results outlined in Section 4, below. This compares favorably with the four-week effort required to tune and determine the optimum deployment configuration for the Storm version.

In optimizing the Streams version of the benchmark, the only significant change made to the application logic was to improve the handling of email body text for the word/paragraph and most frequent word counts. SPL's built-in string handling is based on the C++ `std::string` library. This provides excellent general-purpose, rich string handling functionality, at the expense of some performance overhead. For the somewhat unusual, processor-intensive task of stepping through several hundreds of megabytes of text character by character (to count the number of words and paragraphs and to identify the most frequent word), we converted text to its ASCII representation and operated on it (using standard SPL constructs) as integer data. There was a conversion expense, but it was outweighed by the faster integer operations and the use of built-in SPL functions. In the present implementation, a user-defined list of 8-bit unsigned integers (`list<uint8>`) was used[5].

Streams also has a facility to measure and profile performance and then, based on this profiling, automatically optimize the structure of the deployed application (by "fusing" processing elements to reduce inter-PE overhead). We made use of this for eight of the nine main benchmark workloads[6]. Using this "auto-fusion" yielded reductions in CPU utilization of between 7% and 24%.

It is possible to achieve greater performance (higher throughput and/or lower CPU utilization) by, for example, forcing more aggressive operator fusion or by parallelizing the individual operators that implement the most processor-intensive elements of the application, but the chosen approach was felt to give a good balance of simplicity and ability to optimize further if needed.

3.2 Storm Implementation

The application has been designed using standard Java constructs and Object Oriented Programming principles. Specifically, the application logic is encapsulated within Java classes to maintain a division between the logic functions and underlying Storm constructs. As a result, the same application logic is used across multiple Storm topologies. Another benefit of this scheme is the ability to unit-test the application logic using JUnit. For most cases, we use *localOrShuffle* grouping (which randomly distributes tuples across tasks but first tries to keep tuples local for tasks within the same worker) for all bolts and spouts. In some cases, vanilla *shuffle* grouping was employed as it yielded better performance.

[5] We also tried this byte array based character manipulation in Java but it did not improve performance. Therefore, we did not consider this implementation for Storm.

[6] In the ninth case (the 4-node 100% x4 workload), we configured operator fusion manually. This yielded a 10% reduction in CPU utilization and a 9% increase in throughput.

3.2.1 Storm Application Structure

File Read and Decompress (Spout):

This is implemented as a Spout, which uses `GZIPInputStream` wrapped in an `ObjectInputStream` to read binary email objects and emit them to a downstream operator.

Deserialize (Bolt):

A standard `DatumReader` and `binaryDecoder` from the Avro (ver. 1.7.4.) Java library is used to convert a `byte[]` to an Email object. The fields of this email object are then converted into a Storm tuple and emitted.

Filter (Bolt):

- The `From` String field is scanned for the presence of @enron.com; if it is not found, the entire email is dropped.
- To limit the recipient Strings (To, CC, and BCC fields) to enron.com addresses, the Strings are tokenized into individual addresses and then only enron.com are regenerated into a String using a `StringBuilder`.
- All undesired, non-text characters are removed based on regex matching.

Modify (Bolt):

- Name obfuscation is achieved via `StringUtils` from the Apache Commons library.
- A `HashMap` based algorithm is used to calculate the most frequently occurring word in the email body.

Metrics (Bolt):

- Email, character, word, and paragraph counts are worked out using simple counters. These values are then filled into relevant fields in the email tuple.
- This Bolt emits two different streams: 1) "path" stream which connects it to the serialize bolt. A tuple is emitted on this stream for each input tuple, and 2) "off-path" which connects it to the global metrics bolt (described below). A tuple with fields *<transactionID, #Emails, #Characters, #Words, #Paragraphs>* is emitted after a configurable time window[7] on this stream.

Global Metrics (Bolt):

- For global metrics a single bolt (parallelism of 1 and global grouping on the "off-path" stream) is employed which maintains a HashMap of tuples that it receives from metrics bolts, indexed by *transactionID*. It outputs a consolidated entry once it has achieved the required "quorum" for that *transactionID*. Quorum is equivalent to the lateral parallelism of the metrics bolt. This notion of transactions is used to neutralize any mismatch between tuple arrivals during the same time epoch. For instance, if the metrics bolt parallelism level is two, the global metrics bolt will output a consolidated entry once it has received two tuples from different streams with the same *transactionID*.
- Final values are emitted in the overridden `cleanup()` method of the bolt.

[7] As we discuss in Section 3.3, Storm has no notion of windows so this functionality had to be implemented manually.

Serialize (Bolt):
A standard `DatumWriter` and `binaryEncoder` from the Avro Java library is used to convert email tuple fields into a serialized `byte[]`.

Compress and Write (Bolt):
A `GZIPOutputStream` wrapped in an `ObjectOutputStream` is leveraged to compress and write the final `byte[]` to `/dev/null`.

3.2.2 Storm Benchmark Development and Optimization

In this section, we describe various optimizations that were employed for Storm at both application and platform levels.

3.2.2.1 Application Variants

Other than the standard application that was used for the actual comparison, a number of variants were also initially implemented and then discarded based on their suboptimal performance, such as:

1) **Guaranteed-tuple Processing**
Each tuple emitted by the Spout is anchored and subsequently acknowledged by Bolts down the execution pipeline. This was discarded due to the overhead of *acker* tasks.
2) **Uniform Load-balancing**
Storm using the shuffle and localOrShuffle groupings, randomly distributes tuples across multiple streams. This can be sub-optimal if the tuple size is not uniformly distributed. Consider a scenario with 3 operators, *O1* to *O3*, in which the output of *O1* has to be distributed across *O2* and *O3*. For simplicity assume 4 tuples: *T1* to *T4*, with sizes 1, 1, 10, 10 bytes, respectively. In the worst case, *O2* might always end up with *T3* and *T4* and *O1* with *T1* and *T2*. In essence, based on the application logic, *O2* will end up having to process 10 times more data than *O1*. This is an example of the canonical "balls and bins" problem. To rectify this, each operator was modified to maintain a priority queue of downstream operators and the amount of data (tuple size) it has emitted to each. Each tuple is directly emitted to a downstream operator, which has received the least amount of data so far (a simple priority queue lookup). In essence, this enables sized based load balancing. Based on our previous example, this allows both downstream operators to process the same amount of data, 11 bytes each.
For our workload, this did not improve performance and was thus discarded.
3) **Python Implementation**
The entire application logic was implemented in Python and wrapped in Storm operators. But as Storm launches the Python interpreter within a separate process and connects to it via a pipe, there's a high overhead of object serialization/deserialization to/from Java from/to Python so a native Java approach was used instead.

3.2.2.2 Application Optimizations

1) **Object Re-use**
To minimize the cost of dynamic object creation and garbage collection in Java, extreme care was taken to ensure that objects were re-used across operator invocations. The initial application design based on Java OOP principles

simplified this scheme: all operator invocation invariant objects such as *Lists*, *Maps*, etc. are initialized within the constructor of each logical step. For instance, the *Modification* class, which is in charge of modification tasks, has a function to work out the most frequent word in a String. To this end, it uses a *HashMap*. Rather than allocating a new *HashMap* for each tuple, the same *HashMap* is cleared and re-used.

2) **Memory Efficiency**
   The majority of the application logic revolves around *String* manipulation. In one instance, a set of strings needs to be concatenated. In Java, in general, there are 3 ways to achieve this: direct concatenation, *StringBuffer*, and *StringBuilde*r. All 3 were implemented and evaluated, and *StringBuilder* was found to be more efficient (by an order of magnitude) and thus employed.
3) **Logic Co-location**
   In most cases, care has been taken to ensure that calculations that require iteration through the same data are done at the same time. For instance, for word and paragraph count a simple, single-pass character-by-character comparison was the most efficient implementation as opposed to one that relied on regex matching.
4) **Serialization**
   Storm uses Kryo for serialization due to the inefficiency of the native Java serialization mechanism for high performance stream processing. To ensure that we did not inadvertently use Java serialization, we set `topology.fall.back.on.java.serialization` to false.

3.2.2.3 Platform Optimizations

Several platform optimizations were attempted to improve the application performance. Most of these attempts were not successful, but our results include all of the optimizations that improved the performance of Storm.

1) **Lateral Parallelism**
   The Storm UI exposes a *Capacity* metric that represents the amount of time (0 to 1) in a running window that a Bolt has spent in processing tuples. If this value is close to 1 then the Bolt is a bottleneck in the pipeline and its lateral parallelism needs to be increased[8]. Trying multiple options, optimum parallelism values of 16, 2, and 2 were found for the deserialization, filter, and modification bolts, respectively. This optimization was moderately successful.
2) **JVM Heap Size**
   Each Storm worker is executed within a separate JVM by default with a heap size of 768MB. To ensure that memory was not a bottleneck, the heap size was increased in steps[9]. The heap size was increased until 2048MB but did not yield any difference in performance, thus proving that memory is not a bottleneck for the application.
3) **Queue Length**
   Storm uses two different mechanisms for intra-worker and inter-worker communication. For the former it uses LMAX Disruptor and for the latter, ZeroMQ.
   a. **Intra-worker**
      There are two pairs (send/receive) of queues at thread and executor level (Enumerated in Table 1 with their default values). As queue length can directly affect throughput and queue build-up, we increased their size. The

[8] By passing the required parallelism value to *TopologyBuilder#setBolt*.

[9] By passing the size to the configuration via *Config#put(Config.TOPOLOGY_WORKER_CHILDOPTS, "-Xmx1024m")* for heap size of 1024MB as an example.

Storm documentation also specifies a "high-throughput" configuration (values in third column of Table 1). This did not have any significant effect on performance.

b. Inter-worker

ZeroMQ has a "high water mark" mode that causes the sender to block if the queue reaches a certain threshold. This is turned off by default but can be turned on by setting the value of the configuration parameter *ZMQ_HWM*. Two different values of 1000 and 10000 were tried but both caused the topology to crash. This is in line with anecdotal evidence from the Storm forums where it has been labeled as "bizarre behavior" [23].

4) Dedicated worker nodes

We ran the management services (Storm Nimbus and Zookeeper) on separate nodes so that the worker nodes did not compete with these services for resources and could exclusively use node resources for processing tasks.

Table 1: Message Queue Sizes

Queue	Default	High-throughput
topology.transfer.buffer.size	1024	32
topology.receiever.buffer.size	8	8
topology.executor.send.buffer.size	1024	16384
topology.executor.receive.buffer.size	1024	16384

5) Worker Consolidation

To negate the cost of serialization and inter-worker messaging, multiple workers can be consolidated into a single worker[10]. This was marginally successful and noted to increase throughput and decrease CPU Time.

6) Source Throttling

Another mechanism to negate queue build-up is to throttle the data source (Spout) by simply putting the source to sleep after the emission of a tuple[11]. After a hit and try phase, 1ms was found to be a good sleep interval. Unfortunately, this substantially increased Elapsed Time and CPU Time and decreased the throughput so it was not successful.

It is important to highlight that while Streams worked efficiently almost out of the box, optimizing Storm required an effort spread out over a month.

3.3 Qualitative Differences

Over the course of the work, we noticed the following qualitative differences between the two frameworks:

1. **Application Design:** The Streams Studio simplifies application definition as operators can simply be stitched together according to user requirements. While Eclipse can be used to define a Storm project, it treats a Storm application as a standard Java application while being oblivious to higher level streaming semantics.
2. **Monitoring:** Storm exposes a web interface to a limited set of metrics such as number of tuples ingested and emitted by each bolt but it does not monitor low-level metrics such as CPU utilization, etc. These need to be either manually computed or an external solution such as Ganglia [24] needs to be employed. In

[10] By passing 1 to *Config#setNumWorkers*.

[11] Via *Utils#sleep*.

contrast, both application and low-level metrics can be obtained for a running Streams application by visually inspecting its graph in the Streams Studio. For instance, this interface simplified the collection of CPU times during our measurements. The Streams console and metrics API can also be used to obtain metrics information.

3. **Customization and pre-processing:** Streams supports *mixed mode* application generation in which Perl code is used to generate SPL code at compile time. This is useful for automating complex application design. For instance, the code for a certain operator can be re-used to generate other similar operators. We used this functionality to replicate the SPL code for processing multiple streams. While this "lateral parallelism" can also be controlled to some extent for individual operators in Storm – by directly supplying the parallelism value to the constructor for each bolt/spout – the code itself cannot be automatically customized without resorting to high overhead mechanisms such as Java Reflection and those mechanisms do not work on regions of a graph as they do in Streams.
4. **Windowing:** The SPL runtime contains both tumbling (collective eviction) and sliding (incremental eviction) window operations. These can be used to implement operations that need to be implemented on -- or triggered by -- a time or size based window. Storm does not have the notion of windows; hence, this functionality needs to be hand coded and requires non-trivial spatial and temporal state management and event triggering. Fortunately for Streams users, this sophisticated functionality is built into the framework.
5. **Input/Output:** The default Streams toolkit has source and sink operators for a number of standard I/O types such as files, message queues, databases, etc. The lack of a standard I/O library for Storm forces the practitioner to write code to interface with even simple artifacts such as files.
6. **Punctuation:** Control messages, called punctuation, can be incorporated within tuple streams in the case of Streams. These are useful for transparently bringing down a job once a data source (such as a file) has been drained. It is important to highlight that all of this happens under the hood without any programmer intervention or awareness. On the other hand, for Storm, custom punctuation needs to be implemented that complicates the design of each bolt.
7. **Installation:** Streams ships with an installation manager that installs dependencies and prepares the environment. In contrast, for Storm the practitioner needs to first set up a ZooKeeper cluster and then manually install dependencies, such as ZeroMQ, on all worker nodes, which is quite tedious. This is exacerbated by the fact that ZeroMQ is a native library and others have run into "strange errors" during its installation [25].

4 Evaluation

The comparative evaluation was performed on a six-node cluster with the following specifications and software versions:

- 6-node 3GHz dual-core Xeon cluster (HS21), 2 CPUs per node, 8GB RAM, dual 146GB drives, RHEL 6.2
- Streams v3.1, IBM SDK for LINUX, Java Technology Edition, v6
- Storm v0.8.2, Sun Java™ SE runtime environment 1.6.0_43-b01, HotSpot™ 64-bit server VM 20.14-b01

A Streams instance was deployed across up to four nodes. For the four node cases, the first node executed Streams management services and all four nodes were "workhorses" running Host Controllers. For the 1-node cases, only the first node was used. CPU time was reported using the Stream Studio execution graph by aggregating the CPU time of each Processing Element (PE). Figure 3 shows this deployment visually.

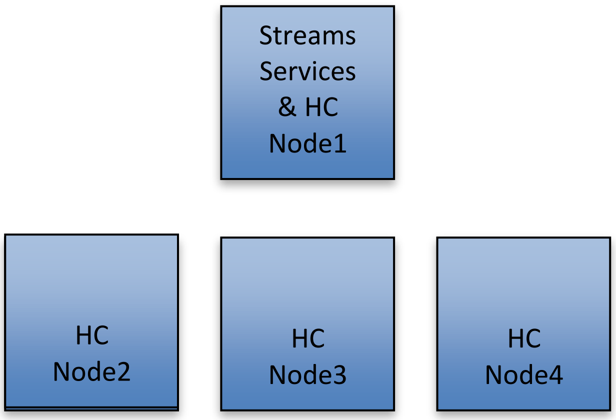


Figure 3: Streams Deployment

The Streams results could have been further improved by using a fifth node for the Streams Services components; however, as the results obtained with this configuration were satisfactory, using a fifth node was not necessary.

For Storm, up to six total nodes were used. The Nimbus daemon and ZooKeeper were deployed on individual nodes and then four nodes were employed for Supervisor daemons (shown in Figure 4). Throughput numbers for Storm were reported from the global metrics bolt while CPU time was looked up in the Linux process tree (using the `ps` command) for each *Worker* JVM. These numbers were then tallied to get a consolidated CPU time figure for the entire job. Finally, as mentioned earlier, *shuffle* and *localOrShuffle* had inconsistent performance. Therefore, we report numbers from whichever option resulted in better performance for Storm.

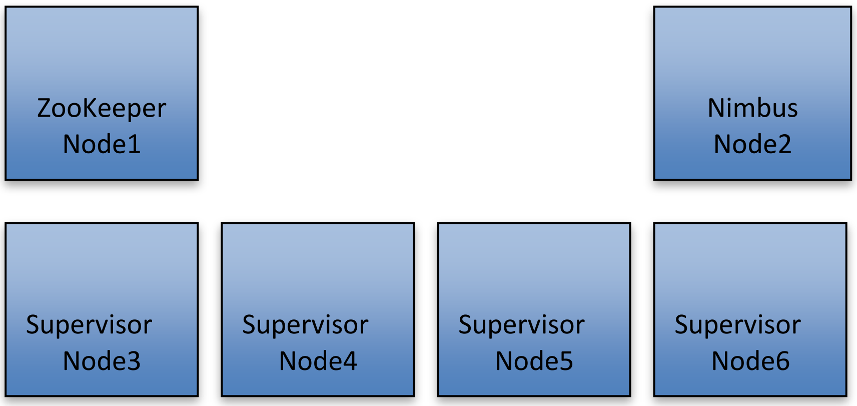


Figure 4: Storm Deployment

We created multiple versions of the dataset containing 25% (approximately), 100%, 200%[12], and 400%[13] of the total emails. Multiple streams were provisioned with their own individual

[12] Two times self-concatenation of the 100% dataset.

[13] Four times self-concatenation of the 100% dataset.

input file, which was stored on an NFS deployment. Furthermore, in the case of Storm, the lateral parallelism of the deserialization, filter, and modification bolts (see Section 3.2.4) was further increased – by a multiplicative factor of 16, 2, and 2, respectively – to ensure that they did not become a bottleneck. For instance, for lateral parallelism of x2 in Figure 5, there were 32, 4, and 4 instances of the deserialization, filter, and modification bolts, respectively, as opposed to 2 instances of the rest of the tasks.

4.1 Application Throughput and CPU Time

All results in this section were measured using the servers, middleware, and application design we have documented. Figure 5 and Figure 6 show the throughput and CPU time, respectively, for a single node deployment. It is clear from the second graph that CPU time consumed by Streams scales linearly with an increase in workload. By contrast, Storm does not exhibit linear scalability in our measured scenarios (this is also apparent in the decline in throughput in the single-node 100% dataset cases and the four-node eight parallel streams cases).

The disparity in performance is such that the measured CPU time consumed by Streams to process one, two or four copies of the 100% dataset is materially lower than the CPU time consumed by Storm to process the same number of copies of the 25% dataset.

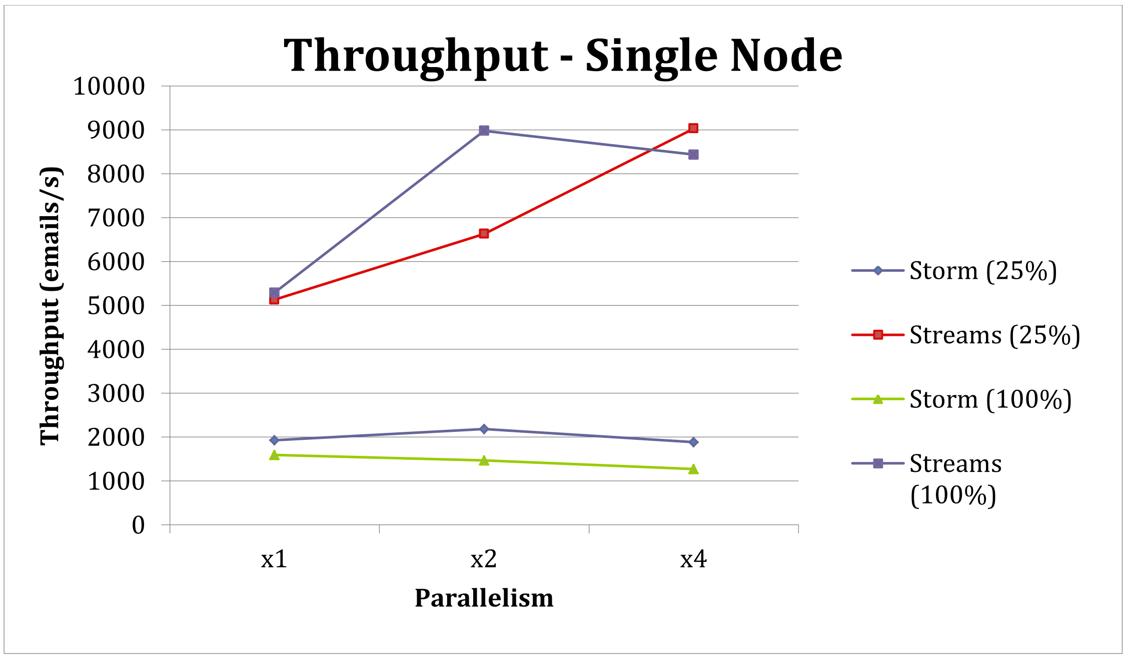


Figure 5: Single Node Throughput

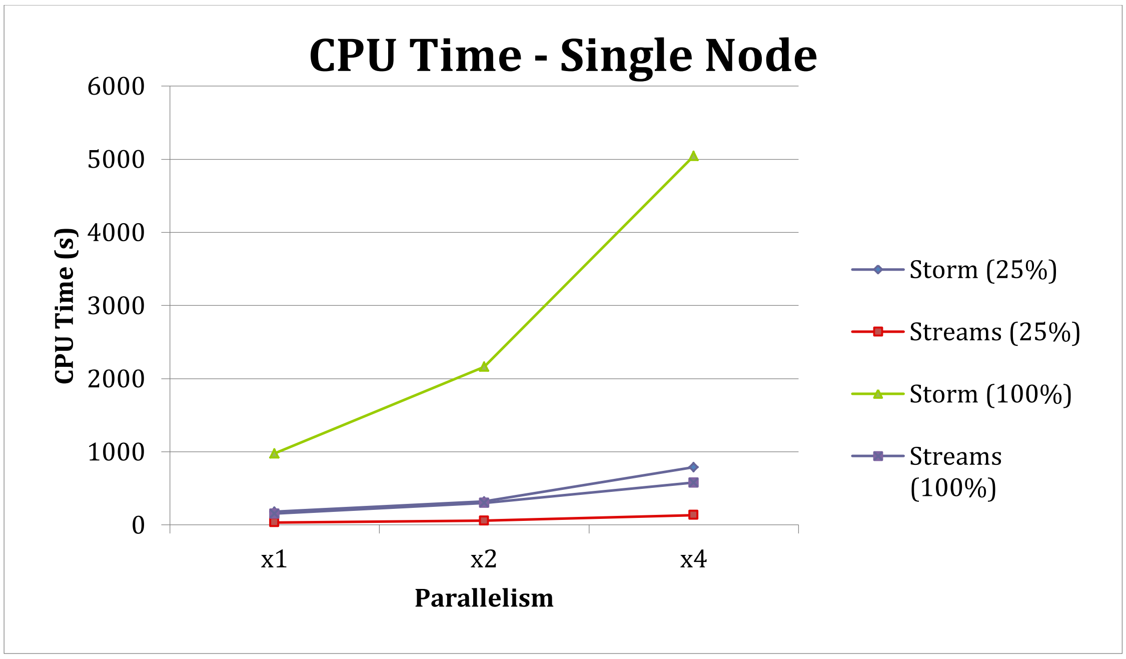


Figure 6: Single Node CPU Time

Figure 7 and Figure 8 show the measured throughput and CPU time for the configuration described above (two management and four worker nodes for Storm, and four nodes in total for Streams). As before, Streams achieves a significantly (4 to 12 times) higher throughput than Storm. In a similar vein, its CPU time consumption remains considerably lower than Storm. Notably, *the CPU time consumed by Streams to process 8 parallel streams of the 200% dataset is considerably less than that consumed by Storm to process 4 parallel streams of the 100% dataset (on both one and four nodes)*.

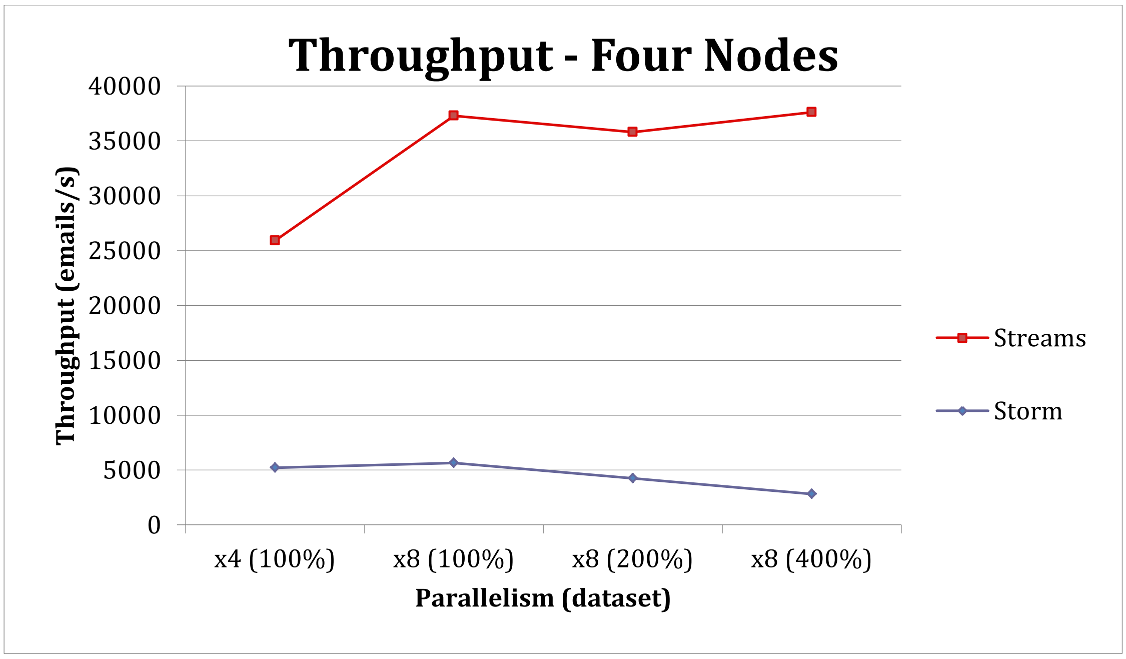


Figure 7: Four Nodes Throughput

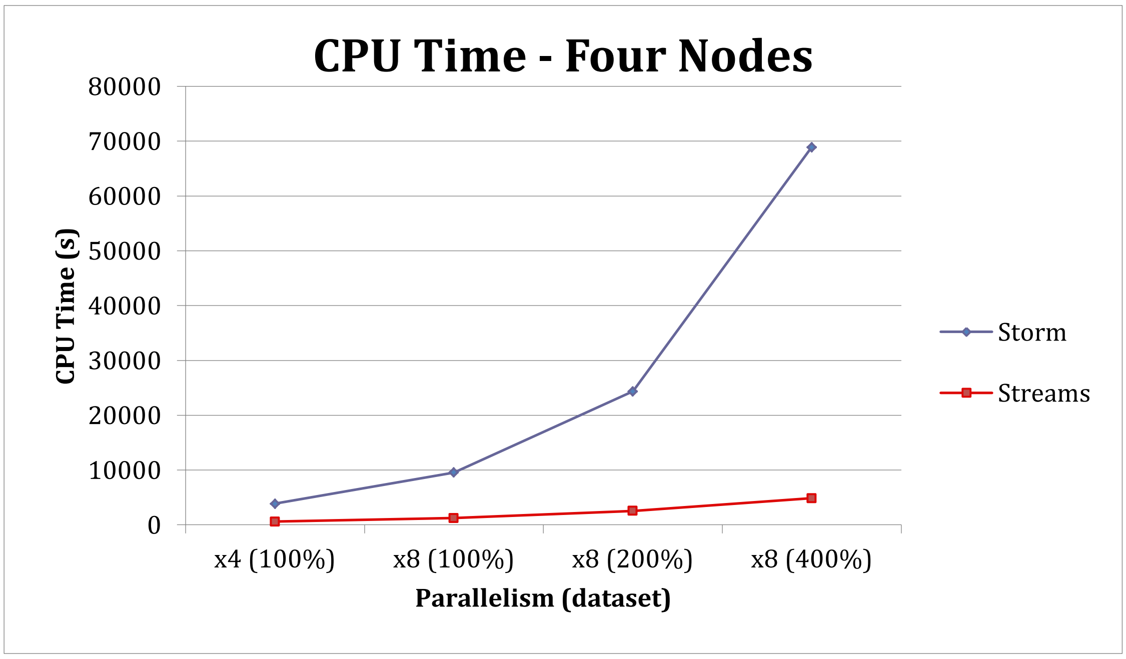


Figure 8: Four Nodes CPU Time

Overall, for the scenarios we have measured with this benchmark, Streams outperforms Storm by **2.6** to **12.3** times in terms of throughput while consuming **5.5** to **14.2** times less CPU time. The reader is referred to Appendix A for detailed results.

4.2 “Barebone” Throughput and CPU Time

To investigate the reasons for the performance gap for Storm, we performed a “barebone” evaluation, which consisted of removing the application logic and pushing raw tuples through Storm, i.e. the topology graph remains the same but the bolts are identity (no-op). Figure 9 shows the comparison of the Storm barebone application versus the complete application in case of Streams for the 100% dataset with 4 parallel streams on a single node.

Even with application logic removed, Storm is unable to match the performance of Streams. This has one main implication: the difference in the performance of both platforms in the previous section is not due to any differences in application implementation (such as language runtime differences) but rather the architecture and implementation of the underlying platforms themselves. Put differently, for the application pattern in this benchmark, *Streams as a platform is far more efficient than Storm.*

To gain more insight into the performance of Storm, we trivialized the benchmark even further by reducing the pipeline to three stages read, count, and write (which is arguably the most basic stream processing application possible). Specifically, the read stage simply reads compressed binary data and pushes it to the count stage, which counts the number of bytes in the tuple. This tuple is then emitted to the write operator, which is a no-op. It is important to highlight that this pipeline involves minimal processing as even compression and serialization have been removed. Figure 10 depicts this processing flow, which we dub the "Restricted Benchmark".

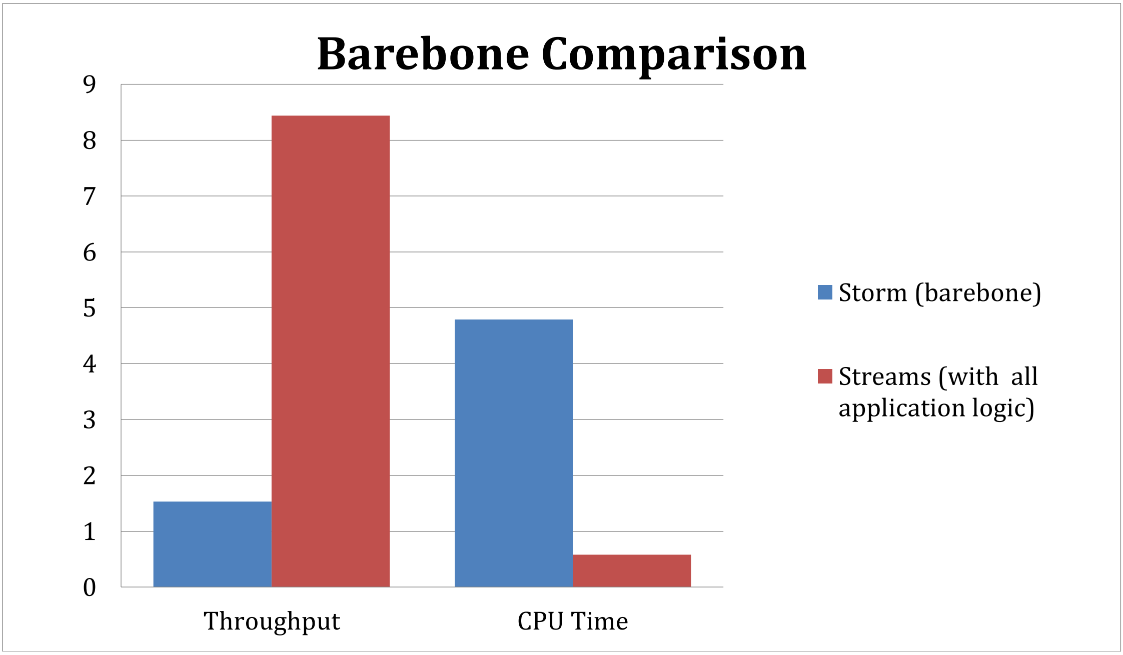


Figure 9: Storm Barebone vs. Streams Complete (Normalized scale)

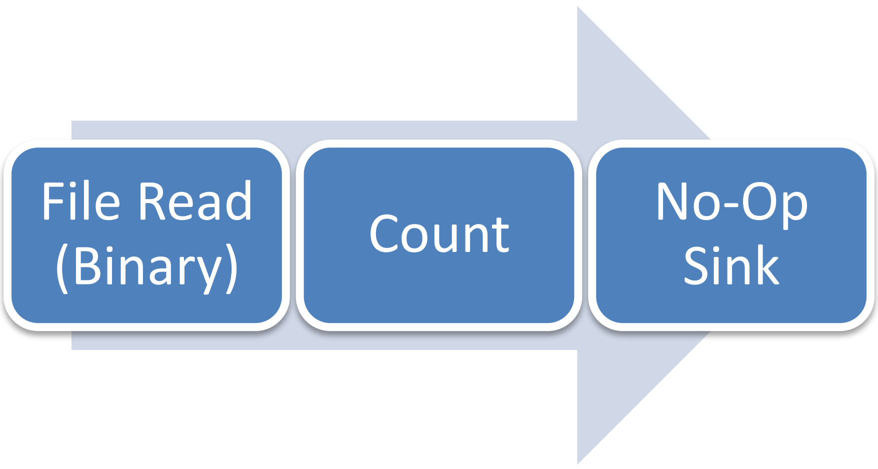


Figure 10: Restricted Benchmark

Figure 11 and Figure 12 present the results of the Restricted Benchmark implemented and run in Streams and Storm. Following the previous results, Streams scales linearly in terms of throughput and outperforms Storm on raw data processing. At the same time, it consumes substantially less CPU time. The results of the Restricted Benchmark further support the better scalability, stability, and resource efficiency of Streams for stream-processing workloads relative to Storm.

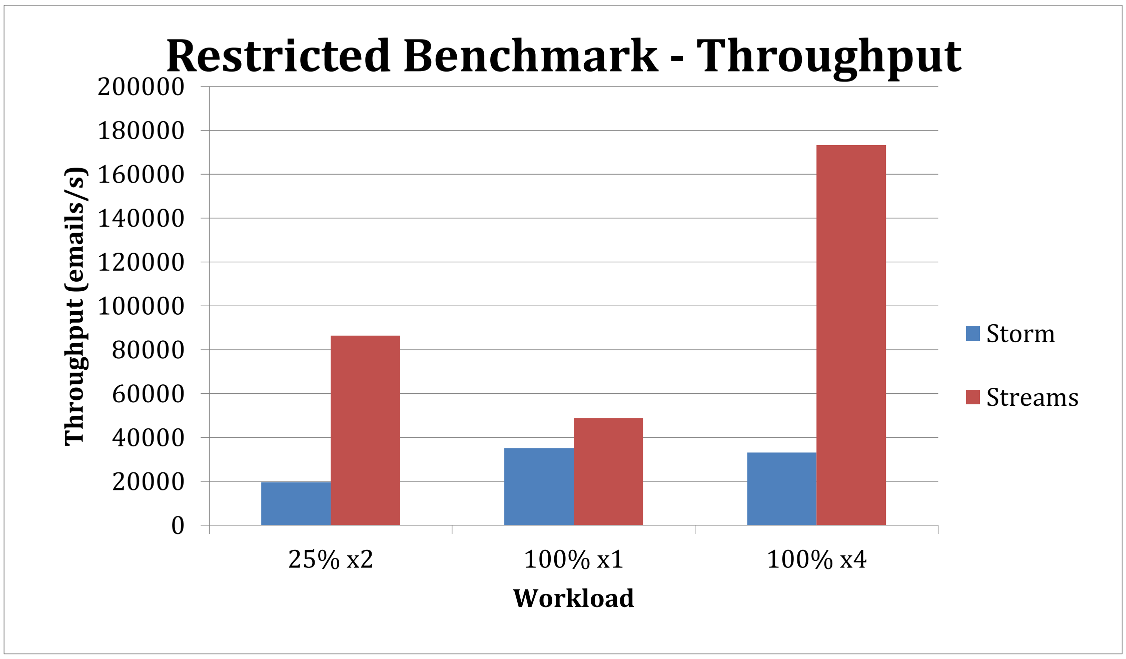


Figure 11: Restricted Benchmark Throughput

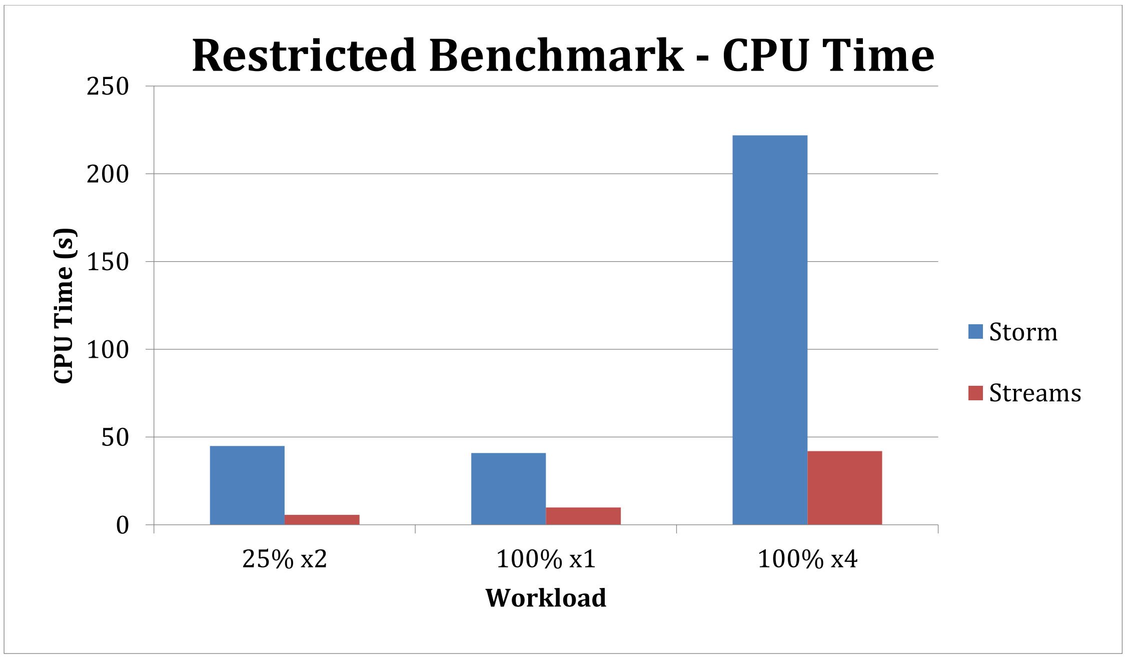


Figure 12: Restricted Benchmark CPU Time

4.3 Micro-benchmark

To diagnose the performance gap for Storm, we conducted an experiment to estimate the impact of tuple transfer on CPU utilization for both platforms. Recognizing that Storm's *localOrShuffle* grouping had a minor impact on CPU utilization compared to fusion in Streams, we chose not to use *localOrShuffle* or fusion in the micro-benchmark. The results of this micro-benchmark provide a baseline that can be used to interpret the performance measurements of the barebone application. To this end, the application used in this micro-benchmark consists of a source operator that sends fixed-size tuples to a sink operator (shown in Figure 13)[14]. In each experiment a tuple is created at initialization time and sent repeatedly from the source to the sink operators without further processing at the application level. This is to ensure that the CPU measurements consist mostly of time spent in transferring the tuples between the operators.

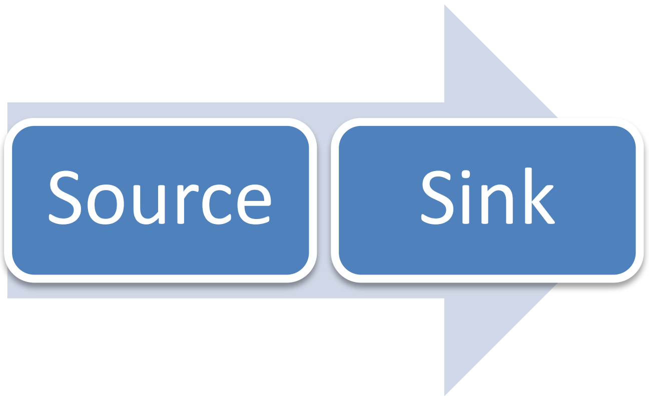


Figure 13: Micro-benchmark

[14] In case of Storm, the source was implemented as a spout and the sink was implemented as a regular bolt.

For the measurements for both frameworks, we deployed the source and sink operators on 2 different nodes. The results of the micro-benchmark are shown in Figure 14 for Storm and Figure 15 for Streams. Each figure shows the CPU usage in percent of a CPU core consumed by the sending and the receiving processes using different tuple sizes.

As expected there is an overhead associated with transferring a tuple in both Storm and Streams. From the figures, it can be seen that it costs more CPU cycles to send a hundred 100 bytes tuples than a single 10k bytes tuple. However this overhead is more pronounced in Storm than in Streams. The figures also show that using Storm we attain 100% CPU-core utilization for both the source and the sink operators when the throughputs are 8MB/s, 52MB/s and 120MB/s for respective tuple sizes of 100, 1k and 10k bytes. In comparison, for those tuple sizes and throughputs, the same application in Streams consumes respectively 70%, 44%, 11% of a CPU core for the source, and 34%, 11%, 19% of a CPU core for the sink.

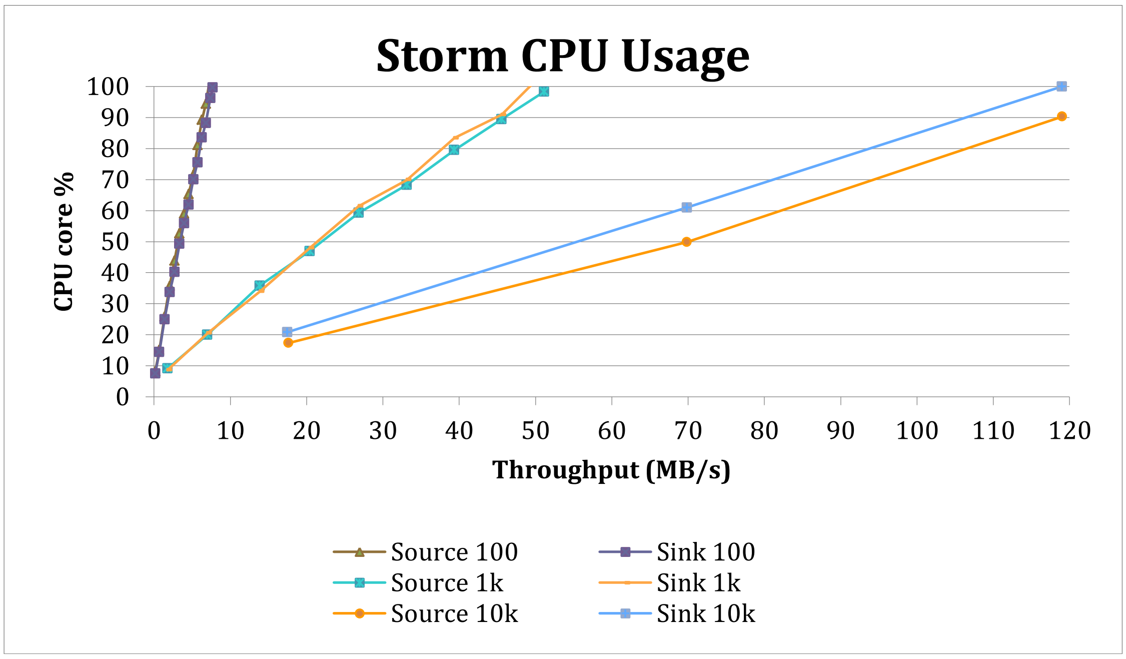


Figure 14: CPU usage as a function of tuple size and tuple rate using Storm

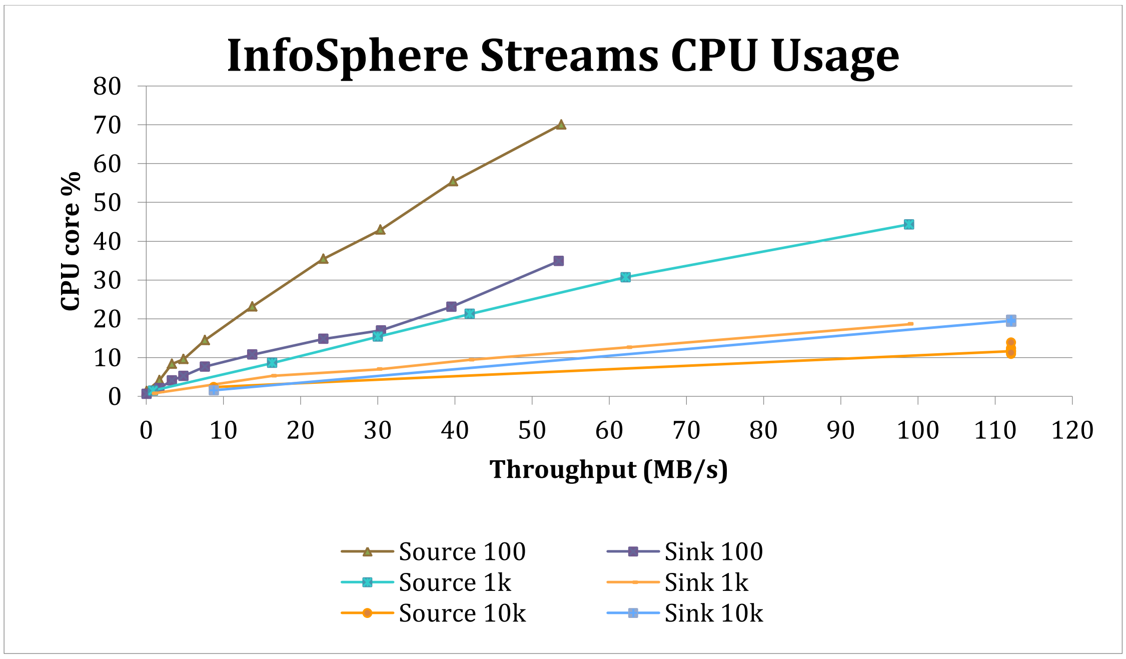


Figure 15: CPU usage as a function of tuple size and tuple rates using Streams

Table 2 summarizes the results of the linear regressions of CPU percentage required per units of 1 MB/s throughput calculated from the results of Figure 14 and Figure 15. The intercept values, not shown in the tables, are about 8% of total CPU core for all Storm measurements and 1.5% for all Streams measurements.

Table 2: CPU percentage per unit of 1MB/s throughput

Payload (bytes)	Storm Source (%)	Storm Sink (%)	Streams Source (%)	Streams Sink (%)
100	13.74	12.95	1.71	0.82
400	4.14	4.08	0.62	0.29
1000	1.95	2.09	0.37	0.2
4000	0.87	1.1	0.19	0.15
10000	0.59	0.88	0.11	0.16

The results in the table can be summarized to say that the message passing function in Streams thus performs between 5 (payload of 100 bytes) and 11 (payload of 10000 bytes) times better than Storm by consuming less CPU. These numbers are consistent with our earlier observations. To put things in perspective, the average size of an Enron email is 2.8KB, distributed according to Table 3. A throughput of 2000 emails per second would thus correspond to 5.6MB/s. More specifically, using the CPU usage per units of 1MB/s in Table 2, weighted by the email size distribution in Table 3, we estimate that the total CPU usage (both Spout and Bolt) to communicate the tuples in Storm is in the range 26% to 32%. Using Streams under the same scenario we obtain a total CPU usage of 4.8% to 5.6% according to the same tables.

Table 3: Enron email size distribution

Email payload range (bytes)	**Average Payload in range (bytes)**	**Percentage of all emails**
0 – 400	398	0.0002
400 – 1000	724	27.2605
1000 – 4000	1972	53.1369
> 4000	9824	19.6024

It is clear that in all cases measured for this benchmark, CPU usage in Storm ramps up very quickly to 100% while Streams keeps its CPU usage in check. Therefore, for even trivial application workloads, the communication sub-layer in Storm consumes most of the CPU. Its performance will further deteriorate with an increase in the application logic as both the application and the underlying Storm engine will compete for CPU. In contrast, the low tuple communication footprint in Streams allows the application logic to consume most of the CPU and exhibit linear scalability.

4.4 Results Synthesis

Our results have shown that Streams outperforms Storm on both the throughput count as well as CPU Time for the benchmarks and measurements conducted. Our initial instinct was that the difference could be accounted to the differences in language used for the runtimes (specifically garbage collection in Java) but deeper analysis has indicated otherwise. In fact, the final application that was used for the comparison can be claimed as a healthy application with respect to garbage collection. For single node execution, the CPU Time overhead of garbage collection varied from 1 to 4% and Elapsed Time varied from 3 to 7%, which (according to an IBM Java Performance Engineer) are the symptoms of a healthy Java application[15]. Therefore, the underlying runtime language differences between Storm (Java) and Streams (C++) play no material part in the differences in performance. Finally, the barebone measurements show that any differences in application logic cost were insignificant, in the context of the overall performance disparity. Our suspicion is that Storm is unable to handle backpressure[16] well, which is in line with anecdotal evidence from other users [26, 27, 28, 29]. This is evinced by some simple analysis, which shows that the throughput of a Storm topology exhibits major fluctuations, indicative of queue build-up. In addition, Streams has been designed to negate the overhead of object typing and serialization, while Storm uses dynamic typing in its tuple definition. Deeper analysis is required to pinpoint the culprit behind the performance gap of Storm and is hence beyond the scope of this work.

4.5 Results Validation

In order to further validate the results, we attempted to run the main benchmark application on a second, and quite different, environment. This was a single-node system, with 8 2.3GHz Xeon cores and 4GB RAM.

[15] Measured by passing *–Xloggc:<log_location> -verbose:gc -XX:+PrintGCDateStamps –XX:+PrintGCDetails* to worker CHILDOPTS

[16] Backpressure is the ability of any pipelined system to deal with queue buildup – in cases where the data source is faster than the sink – by slowing down the source.

Our intention was to run the 100% x4 workload (i.e. four copies of the full dataset with four parallel processing streams). Unfortunately, we were unable to process the 100% dataset using Storm with more than a single processing stream, due to what appeared to be a lack of memory (even the 100% x2 case did not run to completion). By contrast, we encountered no difficulties running the 100% x4 case using Streams.

The only meaningful case (i.e. with parallel processing streams and distribution of workload across multiple Streams PEs or Storm workers) we were able to run on this system was using four copies of the 25% dataset, with four parallel processing streams.

For this case, the CPU time ratio was 4.3, compared with the result of 5.9 we obtained on a single node of the cluster used for the main benchmark testing. We believe this improvement of 27% is due to the availability of additional cores. This result gives us some confidence that our main benchmark results are representative of what may be achievable in other environments.

As noted in Section 3.2.2, we tried many approaches to improve Storm’s performance. The only way we were able to obtain a materially better (from Storm’s perspective) CPU time ratio was to operate without any distribution of workload across multiple workers[17] and to ensure that there was ample spare CPU capacity (i.e. to ensure that CPU utilization is not too high). On this second server, with twice the number of cores, for the 100% x1 case, we were able to reduce the CPU time ratio to between 2.0 and 2.7 (depending upon whether or not the Streams auto-fusion optimization was used). But even in this case, Storm was using the equivalent of **five** cores’ worth of processing capacity, whereas Streams was able to successfully process the same workload using approximately **one** core. This is consistent with our impression that, even with quite limited workloads and without any distributed processing, Storm needs to have significantly greater processor capacity in order to avoid performance degradation.

4.5.1 Storm 0.9

In December 2013 while we were documenting our findings, a new version of Storm was made public [30]. Most notably, the new version optionally allows the user to replace ZeroMQ with Netty, an event-driven communication framework for Java. Some deployments reported up to 2 times more throughput with Netty in comparison to ZeroMQ [31]. Therefore, in the spirit of a thorough comparison, we reran representative workloads from our benchmark with the updated version.

Netty can be enabled by setting `storm.messaging.transport` in the Storm configuration file to “`backtype.storm.messaging.netty.Context`”. For the experiments we used the current stable release of Storm: 0.9.0.1. To first analyze the performance of Netty in isolation, we ran the “Restricted Benchmark” which consists of a simple 3-stage pipeline: source, count, and sink on a 4-node configuration. It is noteworthy that Netty is only used for inter-node communication; therefore a multi-node setup can shed more light on its performance. We found that Storm with Netty is around 1.65 times faster (in terms of throughput) than Storm with ZeroMQ for this setup. Specifically, its throughput went up from 33143 emails/s with ZeroMQ to 54607 emails/s with Netty. It is important to highlight that Storm with Netty is still 3 times slower than Streams for this case, which achieved a throughout of 173355 emails/s.

[17] In practice, this seems equivalent to operating in local mode.

Having analyzed the performance of Storm with Netty in isolation, we next reran the full application benchmark. By default Netty is configured to use a buffer size of 5MB for Storm [31] but in our experiments we found that this value is too high. Specifically, due to the large mismatch in the processing speed of the different components in our pipeline, a large buffer size leads to queue build-up in slower components. As a result, workers run out of JVM heap size. From our experiments, we were able to pinpoint 16KB as a good buffer size, which is in line with the recommendations of others [32]. Furthermore, as Netty is a pure Java solution, it has a higher memory footprint in comparison to ZeroMQ. Therefore, we also increased the default JVM heap size for each worker from 768MB to at least 2GB; otherwise, the workers keep running out of heap space. Finally, our lateral parallelism values of 16, 2, and 2 for the deserialization, filter, and modification bolts improved performance in case of Storm 0.9 as well. As a result, we kept them intact.

Table 4: Storm 0.9 Results

System	Nodes	Dataset	Parallelism	Elapsed time	Throughput (emails/s)	CPU time
Storm 0.8.2	**1**	**100%**	**4**	**21m 29s**	**1,270**	**1h 24m 5s**
Storm 0.9.0.1				**27m 28s**	**988.04**	**1h 28m 2s**
Streams				**3m 13s**	**8,438**	**0h 9m 48s**
Storm 0.8.2	**4**	**100%**	**4**	**5m 14s**	**5,213**	**1h 04m 34s**
Storm 0.9.0.1				**4m 43s**	**5,757**	**1h 04m 56s**
Streams				**1m 43s**	**31,637**	**0h 10m 12s**
Storm 0.8.2	**4**	**100%**	**8**	**9m 34s**	**5,659**	**2h 28m 56s**
Storm 0.9.0.1				**8m 58s**	**6,053**	**2h 18m 56s**
Streams				**1m 43s**	**31,637**	**0h 20m 47s**

Overall, we found that Netty improves throughput by up to 10% but requires many times more memory than ZeroMQ.

Table 4 presents the results of our updated experiments. For reference, it also contains the equivalent numbers for Storm 0.8 and Streams. For a single node deployment, the performance of Storm degraded by 20% from the 0.8 to 0.9. Rows 4—6 present the results for the 100% dataset with a parallelism of degree 4 on a 4-node setup. The results show that the throughput of Storm has increased by 10% with the new version but it is still 5 times slower than Streams. Rows 7—9 list the results for the complete 100% dataset with parallelism 8, Storm 0.9 again improves performance by 7%. It is important to highlight, that while Netty improves throughput over ZeroMQ, conversely, being a pure Java solution, it also increases the memory footprint of the application.

5 Conclusion

The combination of a real-life application and a series of micro-benchmarks enabled us to execute a broad analysis of the performance of Streams and Storm[18]. In the case of the full email application benchmark we performed for this study, Storm CPU time consumption ranged **between 5.5 and 14.2 times** that of Streams while at the same time Streams' throughput ranged **between 2.6 and 12.3 times** that of Storm. These figures do not reflect the further optimization of Streams performance that could have been done by parallelizing individual operators and using dedicated worker nodes (see Section 3.1). Finally, in addition to the stark differences in performance, a number of compelling qualitative advantages of

[18] Naturally, these results, observations and conclusions relate only to the specific workloads measured on the particular system configuration that was used. They are presented to give a perspective on the relative performance of the IBM Infosphere Streams product and Apache Storm but the position with respect to other workloads and in other environments may be different.

Streams, such as an all-encompassing development, operation, and monitoring environment and code customization, were also noted.

Significant efforts were made to tune and improve Storm performance, and to overcome difficulties that were encountered when running larger workloads. Despite extensive analysis and tuning, we were unable to avoid degradation of Storm performance with larger workloads.

Our interpretation of these results leads us to the following conclusions for this benchmark study:

1. Streams' infrastructure for streaming analysis (moving data about, handling backpressure, etc.) significantly outperforms Storm.
2. Streams handles heavy loading much better (which equates to saying that it can make more effective use of available CPU capacity).
3. Storm's performance degradation with meaningful workloads (typical of streaming analysis) suggests that the cost of application logic is unlikely to mask the inefficiency of its infrastructure.

This illustrates the sophisticated and robust engineering of Streams' stream-processing engine; its ability to scale nearly linearly and handle backpressure effectively, while maintaining a low resource usage footprint, is remarkable.

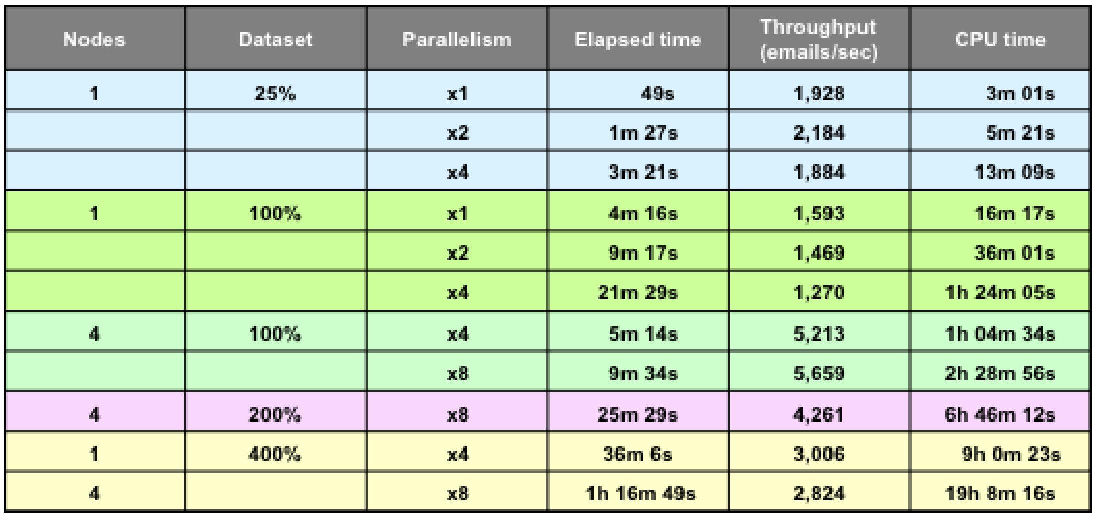
6 References

1. Storm, distributed and fault-tolerant realtime computation. http://storm-project.net/
2. Miyuru Dayarathna and Toyotaro Suzumura. 2013. A performance analysis of System S, S4, and Esper via two level benchmarking. In *Proceedings of the 10th international conference on Quantitative Evaluation of Systems* (QEST'13), Kaustubh Joshi, Markus Siegle, Mariëlle Stoelinga, and Pedro R. D'Argenio (Eds.). Springer-Verlag, Berlin, Heidelberg, 225-240. DOI=10.1007/978-3-642-40196-1_19 http://dx.doi.org/10.1007/978-3-642-40196-1_19
3. Analyzing Behavioral Features for Email Classification Second Conference on Email and Anti-Spam (CEAS 2005) In Second Conference on Email and Anti-Spam (CEAS 2005) (21-22 July 2005) by Steve Martin, Anil Sewani, Blaine Nelson, Karl Chen, Anthony D. Joseph
4. Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. MillWheel: fault-tolerant stream processing at Internet scale. Proc. VLDB Endow. 6, 11 (August 2013), 1033-1044.
5. Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized streams: fault-tolerant streaming computation at scale. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13). ACM, New York, NY, USA, 423-438. DOI=10.1145/2517349.2522737 http://doi.acm.org/10.1145/2517349.2522737
6. Anirban Dasgupta, Kunal Punera, Justin M. Rao, and Xuanhui Wang. 2012. Impact of spam exposure on user engagement. In Proceedings of the 21st USENIX conference on Security symposium (Security'12). USENIX Association, Berkeley, CA, USA, 3-3.
7. Rao, Justin M., and David H. Reiley. 2012. "The Economics of Spam." Journal of Economic Perspectives, 26(3): 87-110.
8. Alain Biem, Eric Bouillet, Hanhua Feng, Anand Ranganathan, Anton Riabov, Olivier Verscheure, Haris Koutsopoulos, and Carlos Moran. 2010. IBM Infosphere Streams for scalable, real-time, intelligent transportation services. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data (SIGMOD '10). ACM, New York, NY, USA, 1093-1104. DOI=10.1145/1807167.1807291 http://doi.acm.org/10.1145/1807167.1807291
9. Olivier Verscheure, Michail Vlachos, Aris Anagnostopoulos, Pascal Frossard, Eric Bouillet, and Philip S. Yu. 2006. Finding "Who Is Talking to Whom" in VoIP Networks via Progressive Stream Clustering. In Proceedings of the Sixth International Conference on Data Mining (ICDM '06). IEEE Computer Society, Washington, DC, USA, 667-677. DOI=10.1109/ICDM.2006.72 http://dx.doi.org/10.1109/ICDM.2006.72
10. Romeo Kienzler, Rémy Bruggmann, Anand Ranganathan, and Nesime Tatbul. 2011. Large-Scale DNA sequence analysis in the cloud: a stream-based approach. In *Proceedings of the 2011 international conference on Parallel Processing - Volume 2* (Euro-Par'11), Michael Alexander, Pasqua D'Ambra, Adam Belloum, George Bosilca, and Mario Cannataro (Eds.), Vol. 2. Springer-Verlag, Berlin, Heidelberg, 467-476. DOI=10.1007/978-3-642-29740-3_52 http://dx.doi.org/10.1007/978-3-642-29740-3_52
11. Biem, A.; Elmegreen, Bruce; Verscheure, O.; Turaga, D.; Andrade, H.; Cornwell, T., "A streaming approach to radio astronomy imaging," *Acoustics Speech and Signal Processing (ICASSP), 2010 IEEE International Conference on* , vol., no., pp.1654,1657,

14-19 March 2010 doi: 10.1109/ICASSP.2010.5495521

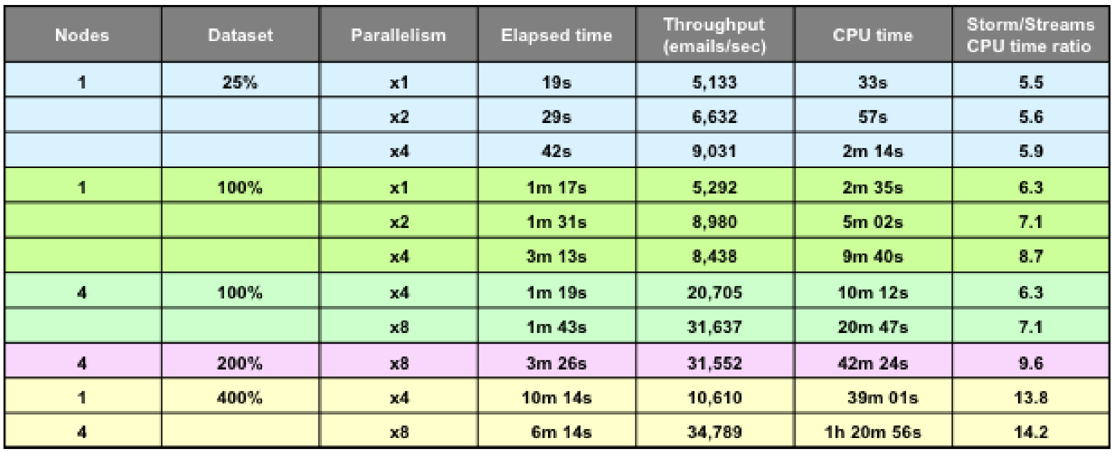
12. Daldorff, L.K.S.; Mohammadi, Siavoush M.; Bergman, J.E.S.; Thide, B.; Biem, A.; Elmegreen, B.; Turaga, D.S.; Verscheure, O.; Puccio, W., "Novel data stream techniques for real time HF radio weather statistics and forecasting," Ionospheric radio Systems and Techniques, 2009. (IRST 2009). The Institution of Engineering and Technology 11th International Conference on , vol., no., pp.1,3, 28-30 April 2009.
13. “Sprint leverages IBM Big Data & Analytics to transform operations”, http://www.youtube.com/watch?v=eg8KSLAZ2HM.
14. Chuck Ballard, Daniel M. Farrell, Mark Lee, Paul D. Stone, Scott Thibault, and Sandra Tucker. 2010. IBM InfoSphere Streams Harnessing Data in Motion. IBM Redbooks. ISBN-10 0738434736
15. Apache ZooKeeper. http://zookeeper.apache.org/
16. LMAX Disruptor. http://lmax-exchange.github.io/disruptor/
17. Distributed Computing Made Simple – zeromq. http://zeromq.org/
18. B. Klimt and Y. Yang. Introducing the Enron corpus. In First Conference on Email and Anti-Spam (CEAS), 2004.
19. Apache Avro. http://avro.apache.org/
20. Protocol Buffer vs Thrift vs Avro. http://ganges.usc.edu/pgroupW/images/a/a9/Serializarion_Framework.pdf
21. V. Metsis, I. Androutsopoulos, and G. Paliouras. Spam filtering with naive bayes - which naive bayes? Third Conference on Email and Anti-Spam (CEAS), 2006.
22. Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", RFC 2045, November 1996.
23. Storm Issues – Replace ZeroMQ with a pure Java solution. https://github.com/nathanmarz/storm/issues/372
24. Ganglia Monitoring System. http://ganglia.sourceforge.net/
25. Storm Wiki – Installing native dependencies. https://github.com/nathanmarz/storm/wiki/Installing-native-dependencies
26. Storm Forum – Regarding Storm internal queues and their configuration. http://grokbase.com/t/gg/storm-user/134kbdp2kv/regarding-storm-internal-queues-and-their-configuration
27. Storm Forum – Internal Storm queues. https://groups.google.com/forum/#!topic/storm-user/9U3RvMv5BYQ/discussion
28. Storm Forum – Spout leaks memory. https://groups.google.com/forum/#!topic/storm-user/lenPQK7k-A4
29. Storm Forum – How to explain periodic variations in throughput. https://groups.google.com/forum/#!topic/storm-user/HNWtrSdILYw
30. Storm 0.9.0 Released -- http://storm.incubator.apache.org/2013/12/08/storm090-released.html
31. Make Storm fly with Netty -- http://yahooeng.tumblr.com/post/64758709722/making-storm-fly-with-netty
32. Hortonworks Technical Preview for Storm -- http://public-repo-1.hortonworks.com/HDP-LABS/Projects/Storm/0.9.0.1/StormTechnicalPreview.pdf

Appendix A



Nodes	Dataset	Parallelism	Elapsed time	Throughput (emails/sec)	CPU time
1	25%	x1	49s	1,928	3m 01s
		x2	1m 27s	2,184	5m 21s
		x4	3m 21s	1,884	13m 09s
1	100%	x1	4m 16s	1,593	16m 17s
		x2	9m 17s	1,469	36m 01s
		x4	21m 29s	1,270	1h 24m 05s
4	100%	x4	5m 14s	5,213	1h 04m 34s
		x8	9m 34s	5,659	2h 28m 56s
4	200%	x8	25m 29s	4,261	6h 46m 12s
1	400%	x4	36m 6s	3,006	9h 0m 23s
4		x8	1h 16m 49s	2,824	19h 8m 16s

Figure 16: Results breakdown for Storm



Nodes	Dataset	Parallelism	Elapsed time	Throughput (emails/sec)	CPU time	Storm/Streams CPU time ratio
1	25%	x1	19s	5,133	33s	5.5
		x2	29s	6,632	57s	5.6
		x4	42s	9,031	2m 14s	5.9
1	100%	x1	1m 17s	5,292	2m 35s	6.3
		x2	1m 31s	8,980	5m 02s	7.1
		x4	3m 13s	8,438	9m 40s	8.7
4	100%	x4	1m 19s	20,705	10m 12s	6.3
		x8	1m 43s	31,637	20m 47s	7.1
4	200%	x8	3m 26s	31,552	42m 24s	9.6
1	400%	x4	10m 14s	10,610	39m 01s	13.8
4		x8	6m 14s	34,789	1h 20m 56s	14.2

Figure 17: Results breakdown for Streams