NTNU – Trondheim
Norwegian University of
Science and Technology

Skyline Computing over multiple Data Streams with a Storm Cluster.

Anders Fredrik Ulvig Kiær

Master of Science in Informatics
Submission date: March 2014
Supervisor: Kjell Bratbergsengen, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

This project aims to design and implement stream processing algorithms for solving the skyline problem over stream data. It includes a short literature study of existing techniques proposed for this topic. Then, the designed algorithms will be implemented using Storm (http://storm-project.net/). The student on this project will become familiar with state-of-the-art technologies for data management and processing in cloud computing or cluster platforms. A prerequisite is good knowledge of a programming languages (preferably Java), and good knowledge of database systems.

Veileder: Kjell Bratbergsengen.

Abstract

Skyline computing has received considerable attention over the last decade. The skyline of a multidimensional point set, consist of interesting points which are not dominated by any other point within base set. Lately the attention has especially been related to computing over data streams. This thesis targets multiple horizontal split streams, using horizontal and vertical scaling provided by Storm, a stream processing framework for cluster. Skylines are incrementally computed when data from streams are received. When multiple layers of skylines are utilized, every increment of a skyline is send onto next layer for further computing. The last layer will contain the final skyline set. Different Storm topologies were proposed, implemented and tested. A discussion of significant observations and an overall conclusion is presented. Some adoptions and optimizations were made towards algorithm completeness to suit an imaged stock exchange setting with limited resources available for a stock trader.

Sammendrag

Skylineberegninger[1] har fått stor oppmerksomhet det siste tiåret. Skyliner fra et sett med flerdimensjonale punkt, består av interessante punkter som ikke er dominert av noe annet punkt fra det opprinnelige settet. I det siste har oppmerksomheten særlig vært knyttet til å beregne skyliner over datastrømmer. Denne avhandlingen sikter på å beregne skyliner over horisontalt delte kilder, ved hjelp av horisontal og vertikal skalering levert av Storm, et strømprosessering rammeverk for maskinklynger. Skyliner beregnes trinnvis når data fra kilder mottas. Når flere lag med prosessering benyttes for å beregne skyliner, blir hvert inkrement av en skyline sendt til neste lag for videre beregning. Det siste laget vil inneholde det ferdige skyline settet. Forskjellige topologier for Storm ble foreslått, implementert og testet. En diskusjon av vesentlige observasjoner og en konklusjon er presentert. Noen tilpasninger og optimaliseringer ble gjort med tanke på algoritmefullstendighet. Det ble gjennomført for å tilpasse en skissert situasjon, hvor en aksjemegler på børsen har begrensede ressurser tilgjengelig for dataprosessering.

[1]Det ble bevisst valgt å bruke det engelske begrepet *skyline*. Det ville kunne oversettes med *horisont*, men det ville nødvendigvis ikke representert beregningen med samme innarbeidede faglige tyngde som *skyline*.

Preface

This master thesis was submitted to the Norwegian University of Science and Technology, NTNU, as a part of the requirements for a masters degree in Informatics.

Acknowledgements

I would like to thank Prof. Kjell Bratbergsengen and Prof. Kjetil Nørvåg from the Department of Computer and Information Science at the NTNU, for their guidance during the work with this thesis. Kjetil guided me at the start of the thesis before he took off for his abroad year of research. Kjell stepped up and guided me through the end. Thanks to you both!

I would like to thank my family and friends for inspiring me and supporting me. An extra thank goes to family and friends lending me some of the needed equipment.

I will also give a special tank to my love for bearing with me during the sometimes hectic work periods.

Contents

List of Figures

List of Tables

Abbreviations

API	Application Programming Interface
CLI	Command Line Interface
CPU	Central Processing Unit
DRPC	Distributed Remote Procedure Call
DSL	Domain Specific Language
GC	Garbage Collector
HDD	Hard Disk Drive
JDK	Java Development Kit
JVM	Java Virtual Machine
MPS	Messages per second
NTNU	Norwegian University of Science and Technology
RAM	Random Acess Memory
SQL	Structed Query Language
UI	User Interface
USB2	Universal Serial Bus 2

1 Introduction

1.1 The Era of Big Data

Rapid advances in data collection and storage technology together with a huge increase of devices containing sensors connected at the edges of networks, made the world generate over 1 ZB of data in 2010, and amazing 2.8 ZB in 2012 and by 2014 we will generate 7 ZB a year [62]. Technology advancement has enabled organizations and government agencies to generate, capture and utilize huge amounts of data.

To handle the increasing flood of data, the trend has been to scale up by increasing storage and processing power in mainframes. As Moores law hit the wall, naturally mainframes did the same. For batch processing, the breakthrough came with Google-filesystem, followed by Hadoop-filesystem, and Hadoop Map-Reduce. Big clusters processing through the big data overnight, with Hadoop, gave an admirable advantage of knowledge. Data that had been considered worthless waste accumulated over time, could now produce useful information. The advantages include showing better search results, more appropriate advertises for products and much more.

It is no big secret that search of knowledge from fresh data is even more valuable. In a competitive market with close to real-time processing of data streams, sometimes two minutes can be too late. E.g. for time-sensitive problem instances like detecting and catching fraud, it is obvious that big data must be used as soon as it streams into your enterprise, in order to maximize its value [17].

1.2 Problem description

With the entrance of high-frequency trading robots to the stock exchange, day trading literally became impossible without help from computers to extract useful information from the stock exchanges' order books. An order book can simply be imagined as a two sided electronic list, to keep track of sellers and buyers interests. At given times, one or more matching engines would run, using the list as input, and fulfil a deal when the interest match.

Most stock exchanges only take a commission when a trade is conducted, allowing robots to flood order books with illegitimate interests that has no other purpose than trying to manipulate the market and deceive humans and other robots. Direct manipulation of stocks is illegal, but the robots are getting away with this behavior. This behavior is rather a political problem.

To face the change, big trading companies put millions of dollars into razor sharp equipment and well educated mathematicians, without facing the resource constraints as individuals or small businesses. The amount of knowledge they are able to extract give them an admirable advantage.

This thesis was done with the following constraints in mind, a stock exchange setting, and a rather limited level of resources available. The problem input was defined as one or multiple data streams, originating from one or multiple stock exchanges, and containing information about stocks and interest changes. The resources available would put an efficient constraint on how much consumption of data that is possible. The data streams are considered endless and may have a higher data rate and throughput, than what available resources can consume. In reality, packages with information would be skipped. In my test cases, the input is tuned or throttled close to the utilized equipment's maximum possible throughput.

With a flood of arriving data one adaptation was utilized. Data considered not useful at the time of arrival are not accumulated for later use. It exist no chance to run over old data a second time, as new and more precious data is already available from the streams. That again imply that we can not get an algorithm output, that can be considered complete.

1.3 The Aim of This Thesis

The aim of this thesis, derived from problem description above, is to introduce low latency skyline computation in a stream environment on an endless stream of generated data. Calculating skylines continuously on the fly instead of calculating it on requests. That will be done on a small cluster using Storm [27], a cluster stream processing framework. This work is hopefully highlighting some of the pitfalls and many challenges on the road.

1.4 Computing Resources

NTNU offered shared access to one of their small clusters. That offer was appreciated but not utilized because of a set of reasons including:

- The ability to utilize the cluster exactly at the time when it was needed. Storm has the ability to run locally for debugging, however some issues is directly related to, or first visible when running distributed.
- The ability to modify all needed software and underlying operative system configuration in any needed way. Including cherry picking a Storm version with specific dependencies.
- Available resources in reach to assembly a personal cluster.

A personal cluster was therefore built in a shared student flat. The cluster network was composed of spare parts originating from participation in organizing computer gatherings. The compute nodes were spare computers both my own and computers supplied by friends and family.

1.5 Outline

Section 1 gives an introduction to this thesis. The remainder is organized as follows: Chapter 2 contains background information on skylines and related work. Section 3 gives a short review on state of the art stream processing frameworks. Section 4 describes core concepts of Storm framework. Section 5 introduces some essential parts of Storms programming interface. Section 6 describes method and implementations. Section 7 present results and a related discussion. Chapter 8 concludes this thesis.

Notices:

- The complete appendix is my work and covers solutions to tasks I ran into during setup, configuring and controlling one small personal Storm cluster. It is meant as a starting point guide for easing replication, and giving possible future projects a smoother start.
- Figures not having a reference are my creation.

2 Background

2.1 Skyline

The Skyline operator has been around for a while in the database research world. It has been a hot research topic over at least the last decade. The problem was known as the maximum vector problem, before [46] introduced it as the skyline operator [52] [46]. Skylines have been found suitable for applications doing multi criteria based optimizations, decision making and finding user preferences [47] [52]. A common example for usage is the stock trader that needs to know which stocks that are worth investing in, based on trading records [47]. The skyline problem was first studied in a centralized way, but as evolution has pushed ahead for storing and processing in a distributed way, distributed approaches have also been covered [52].

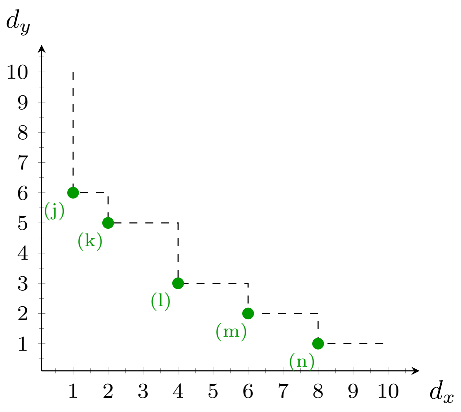
Definition 1 (*The Skyline operator*)**:**
The Skyline operator filters out an interesting set of points from a potential large set of points. A point is interesting if it is not dominated by any other point [46]. A point *p1* is dominating point *p2*, if point *p1* is found to be equal or better than point *p2* in all dimension and is at least better in one of the dimensions [46].

A commonly used problem instance is querying on a hotel distance price relation, a users preference may be to get the cheapest hotels with the shortest distance to the beach [47] [46] [52]. In many cases will the preference for a dimension be a vector containing signs for each dimension that can be used to decide what is better, lower or higher values. It is also possible to implement logic in a domain specified function for handling the comparison and decision. For prototype implementing a skyline is the feature not extremely relevant as it only adds a different constant to the runtime of each *p1* comparison *p2*. So from now on will the thesis stick with the simplification, that the lower values is best. An example skyline is shown in Figure 1 on page 6.

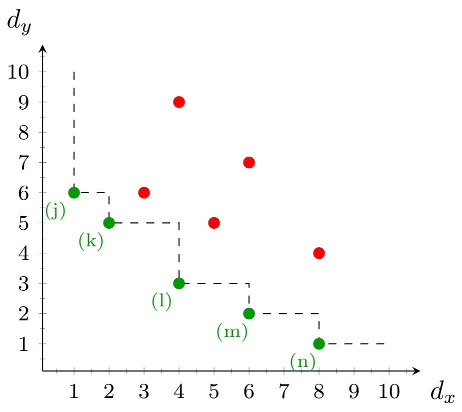
2.1.1 Windows

"A data stream is a real-time, continuous, ordered (implicitly by arrival time or explicitly by timestamp) sequence of items. It is impossible to control the order in which items arrive, nor is it feasible to locally store a stream in its entirety." [51]

As a stream is considered endless, it is obvious that in most cases the problem instance will be constrained to return results from a specific portion of the stream that we are able to keep in memory, called a window. The window could be defined to, e.g. process and return results based on information arrived the last 15 minutes, hour, day or even seconds. Such a window is called a time based window,



(a) Skyline.



(b) Skyline with eliminated points in red.

Figure 1: A two dimensional skyline.

also known as physical window. Windows can also be based on count, or other metrics. A count based window could limit the window to the last ten, hundred, thousand, ten thousand... or even million last received packets. This is also known as a logical window [51].

Windows can be classified with the following domains. The direction of movement and update interval. First the movement, if the window has two fixed endpoints it is defined as a *fixed window.* If it has one fixed and one moving endpoint it would be defined as a *landmark window* and finally if it has two moving endpoints, either forwards or backwards, it is defined as a *sliding window* [51]. Second, the update interval: *Eager* or *lazy.* The eager interval updates the window upon arrival of each new item, also called tuple. The lazy interval is like batching, and induces a jumping window. In the case the interval is larger than the window size, it will result in a non-overlapping series of tumbling windows [51].

When using a query model it is typical to define how often, or at what interval, the operator should run the query over the defined window.

2.1.2 Constrained Skyline

A constrained skyline is calculated with constraints on one or multiple specified dimensions. In a stock trade example a stock trader might want to only obtain skyline points, imitating stocks, with last trading prices between X and Y [47]. Depending on requirements, a subspace can be defined and points failing the constraint can be optimized away. A range specifying a constraint does not need to be closed. A constraint on skyline points may be applied by a complex function that includes usage of knowledge from other data sources, and is therefore not limited to number ranges.

[63, J. Lin and J. Wei] bring up some of the typical assumption connected with computing skylines and highlights constrained skylines as one of them. They also suggest a method for efficient compute constrained skyline points over streams against a count-based sliding window.

2.1.3 Runtime

If the skyline operator is implemented as defined in *Def.1* (on page 5) using nested loop construct, runtime of the d dimensional Skyline will be number of input points n multiplied with points in Skyline m. Runtime will be equal to $O(nm)$, with the compression of points to be done in constant time.

For the special case: one dimensional Skyline, runtime would be trivial. The Skyline m can only hold instances of the current minimum value, that all would have the same numeric value. That taken into account, runtime for this case will be n multiplied with m, and m defined as 1 and n defined as problem input size. So the runtime with one dimension will be equal to $O(n)$.

2.1.4 Optimizations

Many skyline implementations over data streams utilize a count or time based sliding window. Updating a sliding window is expensive and therefore many of the proposed optimizations are directly targeted towards an update of the sliding window as it progresses. One possible way of doing that builds on the idea of memorizing relationships between current skyline points and its successors [59]. It also exist optimizations targeting static datasets, datasets where the entire problem input is available. Existing optimizations strategies towards a static dataset includes usage of; Indexes, R*tree, presorting, divide and conquer (dividing into incomparable partitions), distribution, P2P, usage of filterpoints (thereunder killer tuples), scoring filter and calculations of area of a dominated region. Some of those optimization strategies used for static data can also be utilized over streams.

An optimization for processing points with a high dimensionality over streams, is to vertically split points to temporary disconnect them from dimensions not needed for computing. That will save both bandwidth and increase data locality.

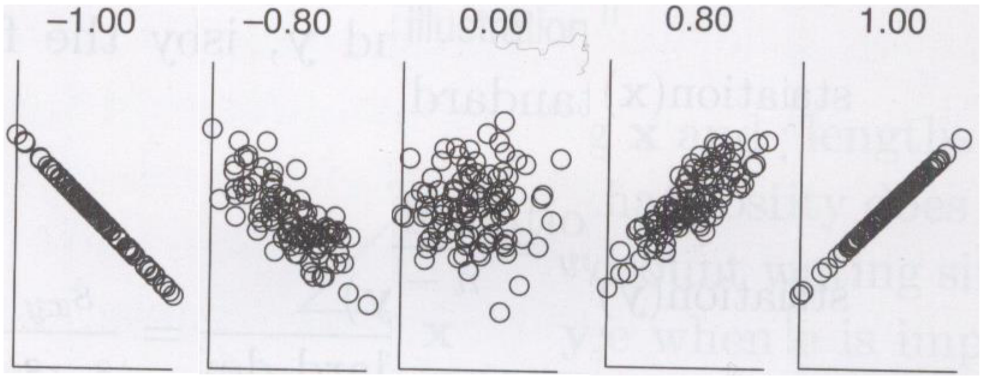


Figure 2: Scatter plots illustrating different correlation.
Stripped down figure from, [57, p.78].

2.1.5 Input Data

There exist many forms of input data. Input data can be based on synthetic data or real data, or a mix. Real datasets, are datasets captured from real world events without any modifications. A modified real dataset is considered to be a synthetic dataset. Synthetic datasets are normally not originating from a real world source. They are typically generated in a sub random way to have a given distribution and comply to a given set of predefined attributes that necessarily would not show up in real world data.

It is common to describe the linear relationships between dimensions in a dataset. That is done statistically with calculating correlation coefficient. A linear coefficient is always in the range -1 to 1. There are more ways to calculate the coefficient depending on the data type. What they all share is that a correlation coefficient of one means that the data points have a perfect correlation. Zero means none correlation found and negative means that the data is anti-correlated [57, p.76-78,375]. Scatter points illustrating different correlations are shown in Figure 2 on page 8.

When big amounts of multidimensional data are processed, they are often subjected to horizontal and vertical split. Horizontal data-split indicates that a dataset with data-points represented as rows, is split into subsets containing complete rows. Vertical split, may be a split on specific dimensions or columns. Vertical split is often done on multi-dimensional data to disconnect dimensions that are not needed for the calculation you want to do on the dataset. Most times an identifier is kept or created, to make it possible to rejoin data-points with its other dimensions at a later point. A mixed split utilize both vertical and horizontal data-splits. The splits are represented graphically in Figure 3 on page 9.

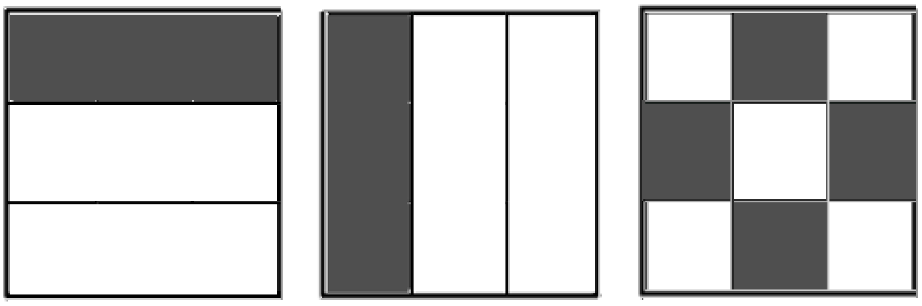


Figure 3: Data splits.
First datasheet is showing a horizontal split on rows. Second a vertical split on columns. Third a mixed split on columns and rows. Grey indicates a possible portion.

2.2 Related work

[52, K. Hose and A. Vlachou] present a great overview over what has been done and some topics that require more attention in research on the skyline operator in highly distributed environments.

[59, Sun et al.] propose BOCS, an algorithm to address continuous skyline queries over multiple distributed data streams. BOCS distributed monitoring architecture is represented in Figure 4. (on page 10). BOCS consists of two parts, a communication protocol named as the algorithm, and a centralized algorithm named GridSky. The BOCS name derives from how the communication protocol is designed. It is *Based On Changes of Skyline.* As the name indicates, only incremental changes also named delta skylines will be sent from remote sites to a coordinator site. The receiving coordinator site preserves the delta skylines and utilizes GridSky to update the global skyline. GridSky is also used on remote sites to do a progressive update of their local skylines.

GridSky is an algorithm for efficiently creating and updating a skyline under a sliding window. It uses an event list to memorize and handle expiration of skyline points, and to replace points with their proper successors. GridSky organizes its data objects in a basic grid. On arrival of new points that grid is utilized for efficiently finding dominated objects and the amount of time it will maximum stay in the window.

The approach of this thesis differs from [59] in multiple ways. With the utilized stream processing framework, a shared nothing can be combined with a shared strategy. BOCS uses a shared nothing architecture. In this thesis a horizontal Central Processing Unit (CPU) scaling over nodes will have a shared nothing ar-

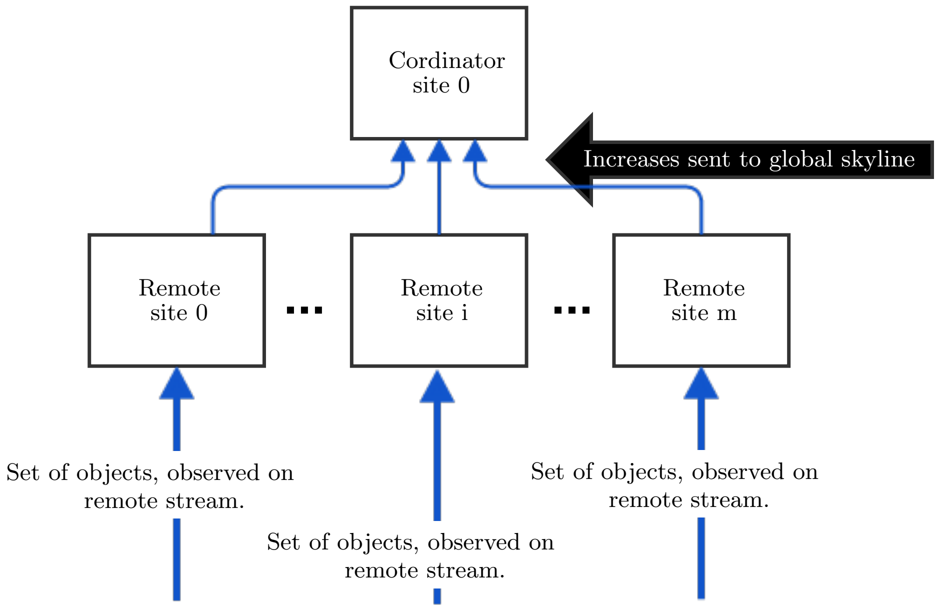

Cordinator
site 0
Increases sent to global skyline
Remote
site 0
Remote
site i
Remote
site m
Set of objects, observed on remote stream.
Set of objects, observed on remote stream.
Set of objects, observed on remote stream.

Figure 4: BOCS distributed monitoring architecture. [59]

chitecture, and vertical CPU scaling on nodes will utilize shared level two caches. This thesis also stands out with its choice of filter usage and the usage of a tumbled landmark window. However, validity of points might be better ensured for general purpose skylines with usage of a sliding window.

3 State of the Art

This part will show some major differences from Storm to other state of the art close to real-time stream processing alternatives. Processing Big Data is not new, and has been done for a while with Hadoop. Therefore Hadoop basics will be covered to some extent to make it clear: Hadoops batch processing, (with Map-Reduce) is very different. A modification of Hadoop can not turn Hadoop in to a high throughput stream-processing framework like Storm.

3.1 S4

”S4 is a general-purpose, distributed, scalable, fault-tolerant, pluggable platform that allows programmers to easily develop applications for processing continuous unbounded streams of data.” [24] S4 might be the closest competitor to Storm. The biggest difference is its lack of support for guaranteed message processing [22]. S4 has the disadvantage of having complex xml to set up its streams, and is also more complex to debug. S4 has automatic load-balancing, but it comes with the disadvantage of opaque task distribution. So overall Storm seems to be superior. We might see a slowly death of S4 as the founder, YAHOO, seems to have taken Storm into production for some of its own loads [20]. The overall activity, and the activity on mailing lists on [25, S4 Project Incubation Status] is not looking too promising.

3.2 Esper

Esper is a component for complex event processing (CEP) and event series analysis. Esper is designed to be highly scalable and fault tolerant. Still the design goal of Esper core is not to scale across distributed Java Virtual Machines (JVM) [53]. Esper can be queried continuously in a Structed Query Language, SQL, like style and contains a lot of the operators we are familiar with from the database domain. Esper leaves it up to the user to feed it with data in a distributed and failsafe way. Esper has a lot of different aggregation-functions that can be done on streams. Where it falls short, Esper was not designed for horizontal scalability on distributed clusters.

3.3 StreamBase

Streambase is a CEP that originate from a high frequency stock trading system for use on the Wall Street, to a fully featured CEP. Streambase is fronting domain specific language, built on the JVM, with a graphical graph based data-flow. It allows extending through an API and is shipped with a pluggable message system supporting guaranteed message passing. StreamBase has cluster scalability, reliability, low latency and high throughput. It can typically process 100 thousand of messages per second. It comes loaded with a great feature set of options, and solutions to typically predefined sets of problems [9] [61].

Compared to Storm, StreamBase is complex software. It is also proprietary and the source code is not publicly available. StreamBases reliability relies on a hot standbys. Hot standby enables a very fast recovery and lower guaranteed latency in the case of failure [61].

3.4 Hadoop

Hadoop is a batch orientated framework for distributed computing, processing massive amounts of data in parallel. Hadoop provides high scalability and reliability for computational problems that can be expressed in the Map-Reduce paradigm [18] [43].

Map-Reduce

The core concepts of Map-Reduce is a mapping phase followed by a reduce phase. The map phase transforms an input data row of key and value to an output list of key/value pairs:

Listing 1: 'The Map [16]'

```
map(key1,value) -> list<key2,value2>
```

For an input row the Map returns a list containing zero or more (key,value) pairs.The output can have a different key from the input, and the same key may have multiple entries [16].

In the reduce phase, "a reduce transform is provided to take all values for a specific key, and generate a new list of the reduced output." [16]

Listing 2: 'The Reduce [16]'

```
reduce(key2, list<value2>) -> list<value3>
```

Map and reduce processes need to be stateless, not depending on any data generated in the same Map-Reduce job to achieve maximum parallelism. It is no feature to control in which order maps or reductions runs. The reduce operation will not happen until all the Maps are done, implying that results are unavailable until the entire mapping phase is finished [18].

The parallel dataflow in a Map-Reduce is excellently shown in Figure 5 on page 13, taken from [48]. The figure shows the enter and exit points for the stored data. The combiner function is an optional function to do partial merging based on keys, having pairs with the same key value to be handled as one group by the reducer. The optional partitioning function is provided to make user-defined partitioning in cases where the general partitioning falls short [48].

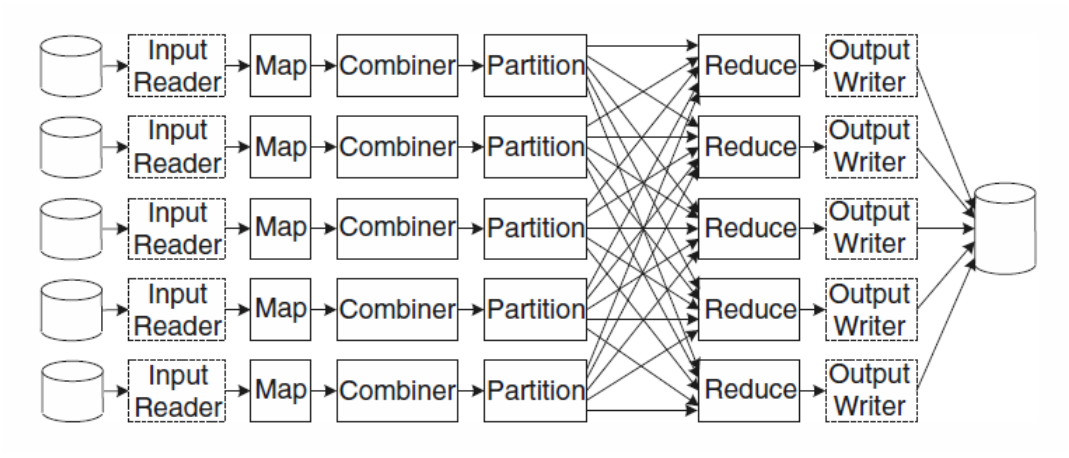


Figure 5: Dataflow in Map-Reduce.
Copied from [48].

Filesystem

Hadoop also provides a distributed file system that stores multiple copies of the data distributed on the compute nodes. It enables Hadoop applications to reliably work with petabytes of data [16].

4 Storm

Storm is a distributed, reliable, fault-tolerant system for processing streams of data [54]. The Storm project is free and got open-sourced 17th September 2011 and is lead by Nathan Marz. It consisted then of approximately 15 thousand lines of code, roughly divided fifty-fifty between Java and Clojure [36] [1]. Storm can be used with many programming languages as it utilizes Apache Thrift for service deployment [6]. Storm also comes with a great and simple API [5].

"Storm exists in the same space as complex event processing systems like Esper, Streambase, and S4. Among these, the most closely comparable system is S4. The biggest difference between Storm and S4 is that Storm guarantees messages will be processed even in the face of failures whereas S4 will sometimes lose messages." [55][2]

Storms message reliability have the ability to guarantee that every tuple emitted

[2] [55] was discovered to be temporary offline or unavailable on 31.03.14

by a spout(source), will be fully processed by topologies when making use of Storms ”at-least-once processing guarantee”. Storms ”at-least-once processing guarantee” tracks every tuple as they travel through the topology. It is one of Storms core mechanisms, and can be imaged done with a tuple tracking tree or a directed acyclic graph (DAG), as a tuple is allowed to anchor to multiple other tuples. One tracking tree will be created for every tuple emitted by spouts. Those tracking trees will receive acknowledgements from bolts. When a tree is fully acknowledged within a given time-limit, the tracker of that tuple will notify the spout so the tuple is not replayed from its source [2]. In the other case, when the tuple is not fully processed or lost, a time-out will occur and the message will be replayed from its source [2].

This basic message reliability achieved from ”at-least-once processing guarantee”, is equal the guarantee from a queue based system [2]. Storm also comes with Trident, a higher level of abstraction that can obtain exactly once processing [2].

Storm comes with no built-in data storage layer. Therefore Storm would typically need an external distributed database like Cassandra [7] or Riak [23], alongside topologies for persistent storage or ability to do a stateless playback from a source (spout) [55]. Storm comes without a storage layer, because it is impossible for one data storage layer to satisfy all different applications that might have different data models and access patterns. Storm is a computing system and not a storage system. ”However, Storm does have some powerful facilities for achieving data locality even when using an external database.” [55]

Storm has many possible use cases like low latency analytic, online machine learning, continuous computation, Distributed Remote Procedure Calls (DRPC), Extract Transform Load (ETL) and more [12] [27]. Some Storm functionality like the DRPC is not covered as it is not considered to be directly relevant for my problem scope.

4.1 Spouts

A spout act as a source, it will typically generate or get data from external sources, and emit them onto one or multiple streams. Where a spout gets its data from is up to the implementation of it. A spout typically implements an interface or extends an abstract implementation to guarantee the basic functionality and the rest is left to the designer. A spout can be designed to read from different queue types like Kestrel(”a simple, distributed message queue system” [42]), RabbitMQ(a reliable enterprise messaging broker [21]) and more. There already exist good example of such implementations. Spouts may read from an external API, like twitters steraming API, a database or simply a file [3] [5]. Spouts may emit zero, one or many tuples on to the desired streams when the next tuple is requested [54]. A spout can be defined in any language, Non-JVM implementations talk to Storm over a JSON-based protocol over stdin and stdout [6].

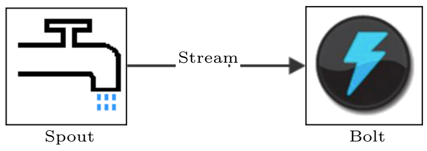


Figure 6: Basic topology with spout stream and bolt.

4.2 Bolts

Logical operations are supposed to be implemented in bolts. A bolt typically implements an interface or extends an abstract implementation to guarantee the basic functionality and the rest is left to the designer. Typical logical functions that could take place in bolts are; filters, aggregations, joins, persistence of state or tuples with the use of databases, functions and so on. A bolt can process any arbitrary number of given input streams and produce output on any arbitrary number of streams, including new streams [5]. When bolts are processing an input tuple from a stream, it can according to implemented logic legally emit zero, one or many tuples on to the desired streams [54]. A bolt can also be defined in any language as Non-JVM implementations talk to Storm over a JSON-based protocol over stdin and stdout [6].

4.3 Topologies

A topology is the top level abstraction of a multi-stage stream computation. A topology expresses the network of spout and bolts, where every edge in the network indicates a bolt subscribing to the output stream of some other spouts or bolts. Storm does allow cycles in topologies. A Storm topology will also run indefinitely, or until manually killed when first deployed [5]. A simple visual understanding of how topologies and bolts will be presented can be found in Figure 6 on page 15.

Storm has multiple different Topology abstractions, one higher level abstractions is Trident. Trident provides ”exactly-once processing, ’transactional’ datastore persistence, and a set of common stream analytics operations” [37] through its interfaces. Trident support aggregations, groupings, filters and functions. TopologyBuilder is a lower level abstraction. It is very basic and has the ability for ”at least once processing” of tuples. TopologyBuilder and Trident is bundled as parts of Storm. It is also worth to mention the transactional topologies that introduced ”exactly once processing” of tuples to Storm. Transactional topologies have become superseded by Trident in Storm 0.8.0, and will therefore be deprecated and scheduled for removal soon [28].

4.4 Streams

Streams in Storm topologies will consist of tuples. A tuple is a named list of values. It will typically contain the values emitted, the tasks ID, the stream ID and the message ID. The stream tuples can carry any object as long as the object has a defined serializer available. Storm uses the Kryo [13] framework for its serialization, but also has the possibility to fallback on Java's slow built in one. "Kryo is a fast and efficient object graph serialization framework for Java." [13] Storm automatically includes serializations for the basic datatypes [5]. For more complex types is it possible to extend Kyro to understand them, write an object description for Kyro and register the extension with Storm.

When setting up Storm streams, how streams are consumed by bolts, Storm has predefined stream groupings for multiple standard applications. This grouping will imply how the bolts running in parallel will share the input streams among them. The bolts can have different patterns of sharing for the different input streams. When assigning an input stream to a bolt it will be identified with the component ID of the producing component [54].

4.5 Clusters

Storm is designed to run on compute clusters. A cluster computing environment uses off the shelf hardware and software connected, often over fast Ethernet connections [49]. The number of computer nodes in a cluster may range from a few to thousands. The nodes will run their own operative system and co-operate in solving larger problems.

For clusters consisting of higher numbers of nodes, the chosen network topology will affect overall performance. The ideal network would have low latency, high-bandwidth, and low overhead protocols. Small clusters typically use star topology (see Figure 7 on p.17). Star topology can be achieved with the use of a single switch. The network can be scaled up to a high number of nodes using tree topology. A downside of tree topology is that it is not suitable for communication intensive applications. The root node will quickly become bottleneck and enforce a different topology. Some switches will let you bind links to obtain fat tree topology (Figure 8 on p.17) and can by that alleviate the problem, but not eliminate it.

Mesh topologies (Figure 9 on p.17) have the benefit of multiple paths giving redundancy. In a full mesh it will be $nodes(nodes - 1)/2$ connections, implying a high cost. A partial mesh benefits from some of the redundancy at a reduced cost, but can not match the benefits of the next topology. There exist cube topologies, but more important hypercube topologies (Figure 10 on p.18). Hypercubes close to eliminate the earlier mentioned bottleneck interconnect problem. In a d-

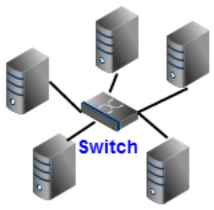


Figure 7: Star network topology.

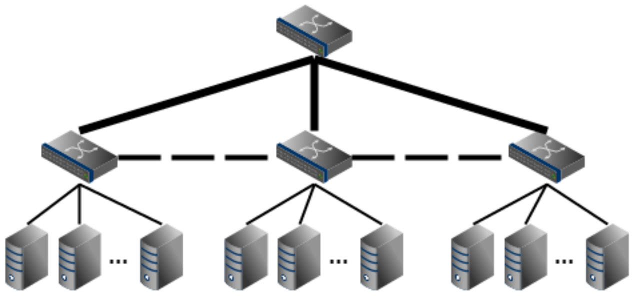


Figure 8: Fat tree network topology.

dimensional hypercube a node will direct connect to d other nodes. The shortest path from a node to another node, will be the Hamming distance. The distance is calculated as the number of bits that differ in the source and the destination address. There are other high performance network topologies related to hypercubes. The hypercube is a toroidal topology with the number of nodes equal two along each dimension. The torus topology is defined as a n-dimensional grid topology connected circular in more dimensions [15]. Both torus and grid based topologies are exploited in high performance clusters.

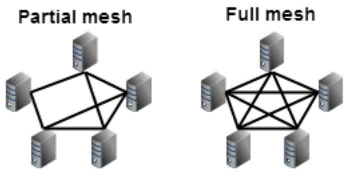


Figure 9: Mesh network topology

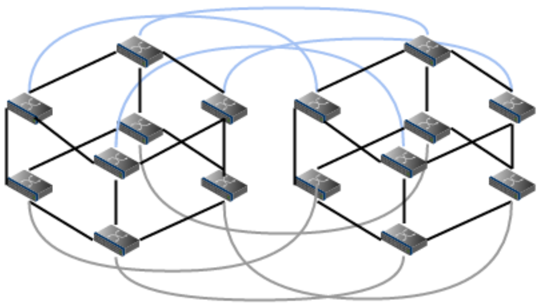


Figure 10: 4D Hypercube network topology

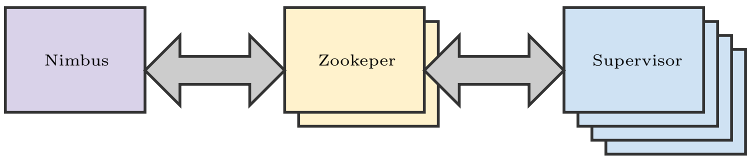


Figure 11: Storm cluster nodes.

4.6 Nodes

The Storm cluster is composed of one master node and worker nodes. The master node runs a daemon called ”Nimbus” which is responsible for distributing code, assigning tasks, and checking for failures. Each worker node runs a daemon called ”Supervisor” which listens for work and starts and stops local worker processes. Nimbus and Supervisor daemons are fail-fast and stateless, which makes them robust, and coordination between them is handled by Apache ZooKeeper [41]. For a visual understanding of the nodes please see Figure 11 on page 18. For a deeper understanding of the fault handling see section 4.9 on page 20.

4.7 Worker breakdown

The supervisor daemon starts and stops worker processes. A worker process is a physical JVM process that executes a subset of all the tasks for the topology [10]. A worker process spawns *Executor*s as threads. Those threads will contain one or more instances of the logic from one type of spouts or bolts. The predefined ratio between Executor and tasks is normally 1:1. For a visual breakdown of the supervisor and workers please see Figure 12 on page 19.

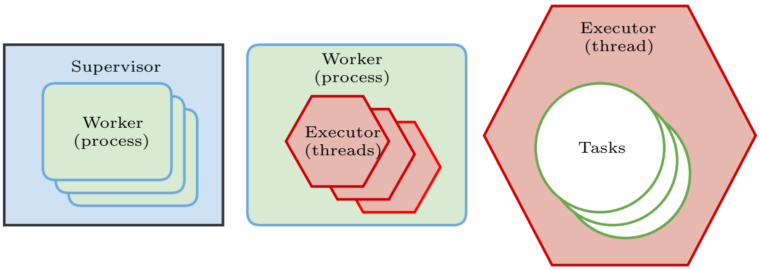


Figure 12: *Supervisor* and *Worker* breakdown.

If the combined parallelism of the topology exceeds the number of allocated workers, each workers *Executor* will execute multiple tasks within a thread. When a sufficient number of workers exist, Storm will try to evenly distribute the tasks (spouts and bolts), across the assigned workers. Figure 12 points out that task is not equal with thread in the Storm architecture. However, in the old Storm architecture task was equal thread, but this complicated the semantics. How? Imagine a stream of words having field grouping performed on it. All equal words will always go to the same bolt so you can easily count them. The problem arise when dynamic scaling of a counting task is needed. It would require creation of another thread and spliting of the existing task. That would not be possible to do dynamically without affecting semantics. The same trouble will also arise with the old model when bolts contain state information [29].

4.8 Workers message passing

Workers are using JØMQ [45] for their direct communication with other workers, over network or socket when on same node. JØMQ is the Java binding for ØMQ [44]. ØMQ is pronunced zeromq and is the name of a socket library that also act as a concurrency framework [44] [45].

Communication within a worker process is different. The communication between threads is done with LMAX Disruptor [11]. That is an awesome high performance inter-thread messaging library, inspired by the principles from more known ring buffer, with some clever modifications on how access to ring buffer is controlled. It also differs from traditional ring buffer in data structure as pointer to the end of the buffer is taken out of the data structure [50]. It is also worth to mention that it avoids locks using semaphores instead. (Compare And Swap/Set (CAS) operations). That gives a admirable boost as it does not include the operative system calls, that leads to context switches and in worst case rescheduling on a different core with plausible lost cache. An explanation of exactly how LMAX Disruptor works can be found in the talk by Martin Thompson and Michael Barker [60]. A

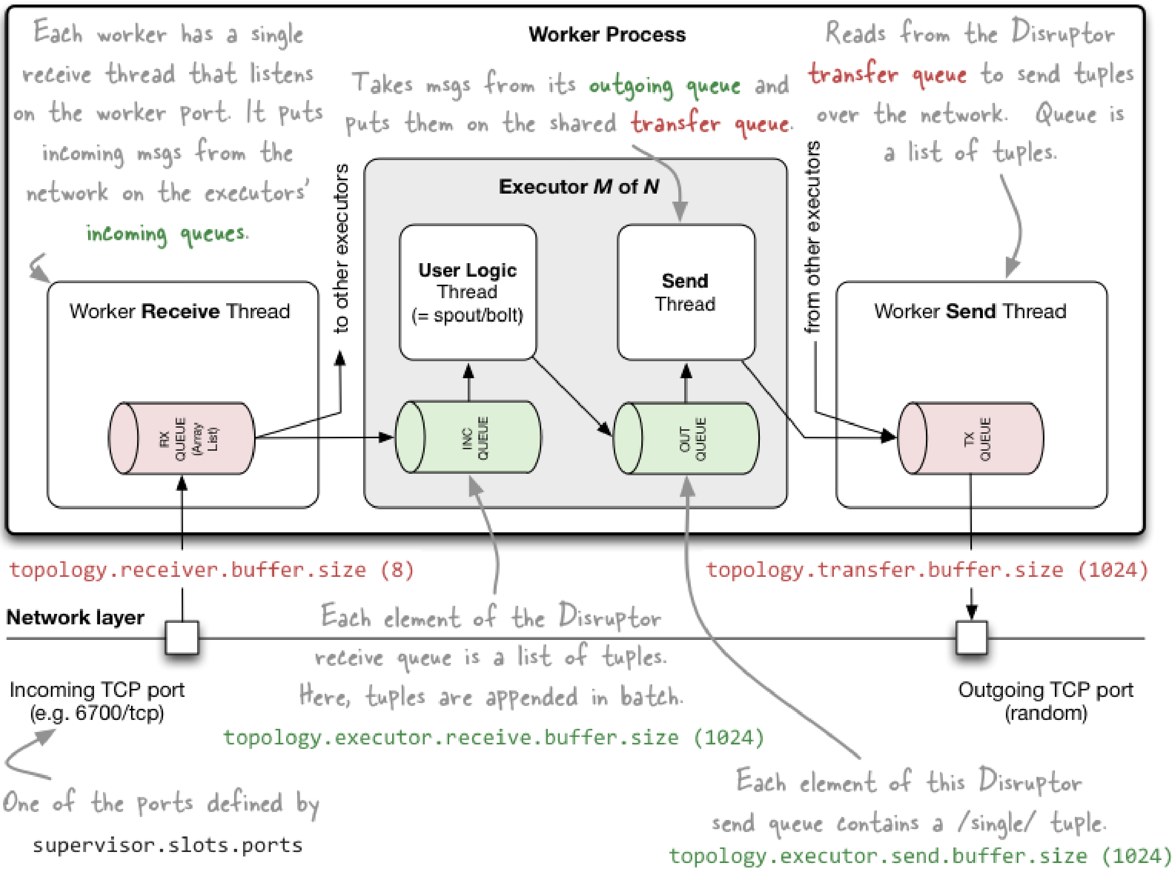

Each worker has a single receive thread that listens on the worker port. It puts incoming msgs from the network on the executors' incoming queues.
Worker Process
Takes msgs from its outgoing queue and puts them on the shared transfer queue.
Reads from the Disruptor transfer queue to send tuples over the network. Queue is a list of tuples.
Executor M of N
User Logic Thread (= spout/bolt)
Send Thread
Worker Receive Thread
Worker Send Thread
to other executors
from other executors
RX QUEUE (Array List)
INC QUEUE
OUT QUEUE
TX QUEUE
topology.receiver.buffer.size (8)
topology.transfer.buffer.size (1024)
Network layer
Each element of the Disruptor receive queue is a list of tuples. Here, tuples are appended in batch.
topology.executor.receive.buffer.size (1024)
Incoming TCP port (e.g. 6700/tcp)
Outgoing TCP port (random)
One of the ports defined by supervisor.slots.ports
Each element of this Disruptor send queue contains a /single/ tuple.
topology.executor.send.buffer.size (1024)

Figure 13: Inter-worker communication.
[56, (c) Michael. G. Noll]

more in depth representation of the overall internal worker massage passing architecture is represented in Figure 13 on page 20. *(Used with approval, thanks Michael.)*

4.9 Reliability

Some of Storms fault-tolerance relies on its use of the architectural fail-fast pattern. Fail-fast implies that execution should fail fast and die visible, instead of trying to recover from an event and potentially reach and unstable state where you might fail slowly [58]. Storm is using this pattern for its components, as it is even harder to debug an unstable distributed problem. Storm is not strictly following the design pattern, as it supplies standard values for parts of it's configuration.

The Storm deamons, ”Nimbus” and ”Supervisor” needs to be run under the watch of an external supervisor service on their representative nodes. If components on a computing node continuously fails and fails to heartbeat, will Storm assign the process to an another available computing node, if none is available will it dictate existing Supervisors to create the dead workers tasks across the cluster as threads [14].

So when workers die will the Storm Supervisor handle the situation. Both the Nimbus and Supervisors are designed to be stateless so when Storm Supervisors resign, the local supervisor service can not manage to bring the Supervisor back online, will Nimbus handle the situation. When Nimbus dies should the local supervisor service bring it back up, should it repeatedly fail will the Storm cluster administrator be dead. In theory, on the first hand do we have a single point of failure here, as it is only one Nimbus instance, with no node to take over the job on a failure. On the other hand is it not posing any big threat to a running cluster, as no worker processes are affected by the death of Nimbus or their Supervisor as the cluster communication and heartbeats are done through the Zookeeper cluster [14].

4.10 Measuring Performance

The Storm User Interface (Storm UI) when running on the Nimbus node, will provide capacity metrics for the topologies bolts and total number of tuples seen on different streams. It will report bolts performance capacity as a percentage of full capacity, calculated by how much time the bolt spent executing tuples, over the last ten minute time interval. If a bolt is close to 100% in the Storm UI, it could be a plausible solution be to increase the parallelism of that bolt [29]. The Storm UI will also provide information of totally acknowledged and transferred tuples on a component level for all running topologies.

4.11 Scalability

Storm is easy to scale, it is just to add new nodes to the cluster and storm will reassign tasks to new machines [29]. You also have options to tweak different parts of a running topology parallelism on the fly [4].

Storm is designed with inherent parallel topologies in mind, making Storm able to process very high amount of messages with low latency. The Storm project has reported Storm to be able to process one million, 100 byte sized messages per second, per node on hardware with 24GB memory and two 12 threads Intel E565@2,4GHz [4]. It is however notable, that Storm Wiki [22, Rationale] page states close to the same for a cluster consisting of ten nodes.

4.12 Roadmap

Storm is on its entry path to become under the umbrella of The Apache Software Foundation. That will most likely increase the ecosystem around Storm and boost its development. It also signals that it is safe to rely on Storm, as Storm most likely will be around for the next years [35].

The upcoming version 0.9.0 has implemented some new interesting features. Ranging from making the messages between workers pluggable, the move from ØMQ to Netty [19], and the added support for blowfish encryption based tuple serialization [39]. Netty is ”an asynchronous event-driven network application framework for rapid development of maintainable high performance protocol servers & clients” [19]. Blowfish is a symmetric-key block cipher that can be used to encrypt data streams.

Other features that might see light is; peer to peer torrent file sharing protocol to spread submitted topologies on cluster. Static swap of running topologies as dynamic swap is rather complex and Storm strives to keep complexity low. One thing that for sure will show up is a resource aware queuing on workers, giving at least the ability to queue on worker node with a specific hardware available. It is also likely to see work being done to increase the parallelism of Nimbus and remove the potential single point of failure [14]. We might see Online Machine Learning Algorithms like what Apache Muppet is for Apache Hadoop for Storm [34]. A Suite of performance benchmarks is also likely to develop and become merged [34].

5 Programming with Storm

An explanation of Storms most basic API is needed for better understanding of the implementations of the skyline-topologies and their representative spouts and bolts.

There are Domain Specific Languages (DSL) for implementing spouts, bolts and topologies in Storm. The focus here will be on Java, as Java has the majority of the interfaces and my implementations are done with Java. The main interfaces of Storm API is the two Java interfaces IRichBolt and IRichSpout, and TopologyBuilder. Storm follows the naming conventions to prefix Java Interfaces with ”I” because Storm also includes a set of base classes that provides the default implementation that can be extended. They can be found in Storm jar under *backtype/storm/topology/base/'*'.java* and are mainly meant as templates that can be extended.

5.1 Spouts

IRichSpout is the main interface for creating spouts with Java, it extends the interface *ISpout* and *IComponent* [33]. The main difference between *IRichSpout* and *ISpout* is that *IRichSpout* adds the method *declareOutputFields*, that serves the purpose of making the user able to declare output stream fields as a part of their implemented spout [38].

A spout run as a *task* inside an *executor*. The *executors* run their assigned *tasks* in an asynchronous loop. Implying that spout logic in *task* also run as an asynchronous loop, also named a tight loop. If *executor* has only one spout task, the execution path in an active spout will be ack(), fail() and at last nextTuple(). All of them inherited from the *ISpout* interface.

ISpout interface has the following characteristics:

Functions are void, and will not return any values on completion.

- *ack()* is taking messageID as a java.lang.Object to identify that tuple with that identifier have been fully processed and should be taken out of the tuple queue.
- *nextTuple()* is not taking any argument. When function is called it is supposed to put one or more tuples into its output streams.
- *fail()* is taking messageID as a java.lang.Object to identify that the tuple with that identifier have failed to be fully processed and should be sutured to outgoing tuple queue.
- *open()* It is called when spout is initialized within worker, it provides an environment for spout. Function can be used to prepare the spout before processing. First argument provides topology configuration. Second argument provides topology context. Third argument provides *SpoutOutputCollector.*
- *close()* Has no argument and is executed when spout is about to shutdown. However it has no guarantee to be called before shutdown because Storm Supervisors 'kill -9' workers. ('kill' sends a signal to a process, the -9 flag sends a non-blocking exit. In a non-blocking exit the kernel may remove the process without informing the process of it.)

Two more functions are worth mentioning: Activating, *activate()* and for deactivating, *deactivate()*. Both functions have no parameters, and can be invoked from Storm UI and Storm Command Line Interface. Those Storm UI commands purpose is to toggle if Storm invoke the nextTuple() function or not. Simply explained those functions in the API are used to arrange access to run code before Storm stops calls to nextTuple() and before it starts calls of nextTuple(). In addition to be invoked on toggling of textitnextTuple() calls, the corresponding logic in spouts declared deactivate will be run on spout creation and activate function will be run before topology starts it initial processing.

5.2 Bolts

IRichBolt is the main interface for creating bolts with Java, it extends the interface *IBolt* and *IComponent* [32]. The main difference between *IRichBolt* and *IBolt* is that *IRichBolt* adds the method *declareOutputFields*, that serves the purpose of making the user able to declare output stream fields as a part of their implemented bolt [38]. In cases where guaranteed message passing is turned off, or when auto acknowledging of received tuples is wanted, can the slightly simpler interface *IBasicBolt* be used. *IBasicBolt* extends *IComponent* like *IRichBolt* but it does not extend *IBolt*.

IBasicBolt and *IRichBolt* require the following functions in bolts implementing one of them; *prepare()*, *execute()* and *cleanup()*. However *IBasicBolt* and *IRichBolt* differ in the arguments. *IBasicBolt* require the *BasicOutputCollector* to be a part of the *execute()* function and *IRichBolt* require its *OutputCollector* to be a part of the *prepare()* function. OutputCollectors does what the name indicates, object is used to emit tuples on to streams.

Bolts as spouts, run as one or multiple *task*s inside an *executor*. *Executors* run their assigned *task*s in an asynchronous loop. Implying that logic in the *task* run as asynchronous loop.

IRichBolt and IBasicBolt have the following in common:

Functions are void, and will not return any values on completion.

- *prepare()*; It is called when bolt is initialized within a worker, it provides bolt with an environment. Function can be used to prepare bolt for processing. First argument provides topology config. Second argument provides topology context.
- *execute()*; It is called every time bolt is processing a tuple. Received tuple is first argument of the function.
- *cleanup()*; Has no argument and is executed when bolt is about to shutdown. However it has no guarantee to be called before shutdown because Storm Supervisors 'kill -9' workers.

5.3 Topologies

Storm has multiple different abstraction levels. *TopologyBuilder* is the most basic one and will be covered to some extent here. The *TopologyBuilder* class exposes the Java API to ease process of creating a Thrift structure. Thrift is a interface definition language that is used to define and create services. Thrift is capable of creating a complete stack for servers and client services [8]. More details of Storms Thrift usage can be found in [36]. The *TopologyBuilder* hide the complexity of the underlying Thrift structure and thereby greatly eases the process of creating topologies [40].

Stream groupings available for use when crating stream networks in topologies are:

1. **Shuffle grouping**: Tuples are randomly distributed across the bolt's tasks in a way such that each bolt is guaranteed an equal number of tuples.
2. **Fields grouping**: The stream is partitioned by the fields specified in the grouping. For example, if the stream is grouped by the "user-id" field, tuples with the same "user-id" will always go to the same task, but tuples with different "user-id"'s may go to different tasks.
3. **All grouping**: The stream is replicated across all the bolt's tasks. Use this grouping with care.
4. **Global grouping**: The entire stream goes to one single task. Specifically, it goes to the task with the lowest id.
5. **None grouping**: Specifies that it is not important how the stream is grouped. Currently, none groupings are equivalent to shuffle groupings. Eventually though, Storm will push down bolts with none groupings to execute in the same thread as the bolt or spout they subscribe from (when possible).
6. **Direct grouping**: This is a special kind of grouping. A stream grouped this way means that producer of tuple decides which task of the consumer will receive this tuple. Direct groupings can only be declared on streams that have been declared as direct streams. Tuples emitted to a direct stream must be emitted using one of the emitDirect methods. A bolt can get task ids of its consumers by either using the provided TopologyContext or by keeping track of the output of the emit method in OutputCollector (which returns task id that tuple was sent to).
7. **Local or shuffle grouping**: If the target bolt has one or more tasks in the same worker process, tuples will be shuffled to just those in-process tasks. Otherwise, this acts like a normal shuffle grouping.

[10, List and explanation, copied with minor modifications from Consepts, Storm Wiki.]

5.3.1 Configuration

Storm has an extensive set of available configuration options for different components. When guide in the appendix is used to set up a Storm cluster, those options can be defined on nodes in the file *"/home/stormuser/storm-0.8.2/conf/storm.yaml"*.

A more practical way that allows options to be topology bound instead of node specific, is to set options from within Java. That can be done using a plain Java Map that is attached when submitting topology to the cluster. The same way can be utilized to submit options from a command line interface. I found it preferable to keep settings local to topology and attaching plain Java Map on submit of topologies to the cluster.

The different ways to set options are illustrated below.

Listing 3: 'topology config example code'

```java
// The Config class is a Java Class that extends java.lang.HashMap.
Config configObject = new Config();

// Some configuration options have their own setters.
// Enable debug output. Bolts and spouts will log eg. every tuppel they
    emit.
configObject.setDebug(true);

// The Java Map put way, with one of my Enum types.
// What type of generator shall be used. /* UNIFORM, ANTICORRELATED,
    CORRELATED */
configObject.put(CustomConfig.GeneratorType,
    DistributionType.ANTICORRELATED.getId());

// The Java Map put way, with a value.
configObject.put(conf.TOPOLOGY_EXECUTOR_RECEIVE_BUFFER_SIZE,1024);

// Make sure to attach the created configObject when submitting the
    topology.
// StormSubmitter.submitTopology(topologyName, configObject,
    builder.createTopology());
```

There are numerous possible configuration options for Storm. This part will highlight some options used, the rest is found in [31, Storm api - Config]. Storm ignores any unknown configuration options. Storm encourage users to extend available configuration options to suit their personal needs, as the configuration can be reached from within spout and bolts. I made use of this feature and cover those extended configuration options as well.

Storm configuration options:

1. **Config.TOPOLOGY_ACKER_EXECUTORS** This option sets the number of executers to spawn for acknowledging tuples in the topology. Setting it to 0 will disable reliability and tuples will be instantly acknowledged when they leave spouts.
2. **configObject.setDebug(value);** Vale is False or True. This option toggle logging of emitted messages in Storm topology.
3. **configObject.setMaxSpoutPending(*value*);** Sets the maximum number of pending tuples for one spout. This option has no effect when reliability is turned off.
4. **configObject.setNumWorkers(*value*);** Set number of worker processes to utilize for this topology.
5. **Config.TOPOLOGY_TICK_TUPLE_FREQ_SECS** Set the frequency in seconds, for how often a component shall receive a tick tuple from the "__system" component and "__tick" stream. Option is meant to be component-specific and can be set with creating a configuration object in components *getComponentConfiguration* method. After creating the configObject and setting this option, return the configObject.
6. **Config.TOPOLOGY_FALL_BACK_ON_JAVA_SERIALIZATION**: In a production environment it should be set false. Using the standard Java serialization is extremely expensive for tuples. It is made available only for prototyping.
7. **Config.TOPOLOGY_SKIP_MISSING_KRYO_REGISTRATIONS**: When set to true, Storm will ignore Kryo serializer registrations that ain't available on the JVM classpath.
8. **configObject.registerSerialization(double[].class);** This object setter can be used to register Kryo serialization classes. The parameter given, *double[].class* register a pre-implemented Kryo serialization for an array of the primitive double datatype.
9. **Config.TOPOLOGY_WORKER_CHILDOPTS**: Topology specific, JVM processes started by the Storm Worker will have the given string attached as a command line argument when JVM process starts. This will come in addition to the *Config.WORKER_CHILDOPTS*.
10. **Config.WORKER_CHILDOPTS**: Workers started by this Storm Supervisor will have the given string attached as command line argument to the JVM process when started.
11. **Config.TOPOLOGY_EXECUTOR_RECEIVE_BUFFER_SIZE**: Changing executor's in queue size. Maximum number of tuples held by executor awaiting for local processing of task. Takes a value of the form 2^n. A to high value typically will lead to starvation of heartbeats, as heartbeat is queued as normal tuples.
12. **Config.TOPOLOGY_EXECUTOR_SEND_BUFFER_SIZE**: Changing the executor's out queue size. Maximum number of tuples processed by task, awaiting transfer to shared network transfer queue. Takes a value of the form 2^n.

13. **Config.NIMBUS_TASK_TIMEOUT_SECS**: The time in seconds before Nimbus time out a not answering task.
14. **Config.ZMQ_HWM**: Worker to worker communication. ØMQ Buffer size in items before it takes action eg. blocking.
15. **Config.ZMQ_THREADS**: Number of ØMQ IO-threads that should be used by the ØMQ in each worker process.
16. **Config.ZMQ_LINGER_MILLIS**: Setting the timeout for ØMQ in milliseconds.

Extended configuration options:

1. **CustomConfig.DimensionMaxValue**: Highest value for a dimension in the point. Only relevant for spouts in GenerateDatasetTopology.
2. **CustomConfig.PointDimensions**: Indicates which dataset to load, and therefore also how many dimensions a point in the dataset have. Used by all components that need to know dataset dimensionality.
3. **CustomConfig.GeneratorType**: Indicates the distribution to be used. It is used by the spouts reading a dataset or generating a dataset.
4. **CustomConfig.ToggleSleepValue**: This is used as a throttle for *BinFileReadSpout*. It is a basic counter that adds one each time the spout run *nextTuple()*. When it reaches the limit, spout sleeps for a short period of time and sets counter back to zero. Default value when not specified is 10 times. The value -1 disables the throttle.
5. **CustomConfig.ToggleSleepTime**: Set *BinFileReadSpout*s sleep time in milliseconds, default: 1 ms when *ToggleSleepValue* is enabled, and this configuration option is unset.

The relevant code can be found in the supplied jar under the *source* folder, in the package *no.stud.util*.

5.3.2 Tick Tuples

This subsection introduces Tick tuples to hopefully ease the understanding of how the *SimpleSkylineBolt* (described in section 6.4.2 on page 37) outputs the cardinality of bolt's skyline on a 10 second interval.

Tick tuples are tuples generated on time based intervals to have components do a certain task on a fixed interval. The time interval is set in seconds with configuration option *TOPOLOGY_TICK_TUPLE_FREQ_SECS*. The configuration option is local to component, implying that different bolts have the possibility of having different time intervals. An example is shown below:

Listing 4: 'getComponentConfiguration'

```
        public Map<String, Object> getComponentConfiguration() {
                Config configObject = new Config();
                configObject.put(Config.TOPOLOGY_TICK_TUPLE_FREQ_SECS, 10);
                return configObject;
        }
```

Tick tuples originate from "__system" component and should be received from "__tick" stream. One way to check if received tuple is a Tick tuple is shown below:

Listing 5: 'Tick tuple check'

```
// An example bolt execute method:
public void execute(Tuple tuple, BasicOutputCollector collector) {
        // Check if tick tuple:
        if(tuple.getSourceComponent().equals(Constants.SYSTEM_COMPONENT_ID)
            &&
            tuple.getSourceStreamId().equals(Constants.SYSTEM_TICK_STREAM_ID))
            {
                \\ It is a Tick tuple.
        } else {
                \\ It is a normal tuple.
        } // End if
} // End function
```

Testing of tuples can be further simplified. Implementing the logical expressions in a method returning one boolean value will result in a test like e.g: "if(isTickTuple(tuple))...".

5.3.3 Building Topologies

When building topologies the first step is to decide on which topology abstraction to use. As mentioned, in this case the *TopologyBuilder* abstraction will be used. To build a topology with two spouts, emitting onto two streams, and one bolt consuming those streams will the following be needed:

An instance of the *TopologyBuilder*, lets name it *builder*. That is done in line #1 in the *topology example code* below.

Next we need to create two spouts and give them a unique ID. The ID is referenced by bolts that consume the output of those spouts. Spouts are give IDs *SpoutOne* and *SpoutTwo*. That is done in lines #3 and #4 with the first argument of *setSpout*. The *setSpout* functions second parameter is a class implementing a spout interface, with other words an implementation of a spout. The third parameter should be a number that indicates the parallelism of that spout in terms of how many executors to allocate for it. The *builder.setSpout()* function returns the declared spout-object so additional settings for the spout can be submitted directly to the returned object. Appending *.setNumTasks(n)* on the object will give spout a total of n tasks across topology.

In line #3 a spout is created with the component ID *SpoutOne*, having one initial executor running one task. Line #4 creates a spout with the component ID *SpoutTwo*, having four initial executors running 128 tasks spread on executors. It is time for a quick recap; Task parallelism is static for the lifetime of a topology, and thereby can not be changed on the fly with the Storm rebalance command. In the example code, a rebalance could be used to adjust the number of executors for *SpoutTwo* to a number between one and 128. The same could not be done for SpoutOne as it only has one task. When parallelism or task parallelism hints are not set, both will be set to one.

Line # 6 adds a bolt to the topology. The *setBolt* function has the same characteristic for its arguments as the *setSpout* function. First is component ID, second a class implementing one of the available bolt interfaces, followed by the executor parallelism hint. The *setBolt* function returns the declared bolt-object making it easy to set additional options for the declared bolt on the same line of code. In addition to define bolts IDs and parallelism hints when declaring bolts, it is usual to declare what stream(s) they should subscribe to and consume.

BoltOne declared on line #6 is initially assigned one executor, but has two tasks. It is assigned to consume the default streams from *SpoutOne* and *SpoutTwo*. The stream groupings are described in section 5.3 on page 25. In this example *Shuffle grouping* is used, and the two tasks will get an equal number of tuples from both spout streams. Imagine the bolt is programmed to sum up all seen tuples and output the final sum to standard output once every 10 second. In this case with two bolt tasks it would be one executor running those sequential inside an executor. Two tasks will therefore lead to two partial sums that have to be added together for a total sum. In cases requiring a total sum, there are two obvious ways to reach it; Either set the task parallelism to one, or add additional bolt that subscribe to the output of *BoltOne* with taskParallelism equal one. In last case *BoltOne* would also require modification to output its partial results to a stream instead of standard out.

Listing 6: 'Topology example code'

```
1   TopologyBuilder builder = new TopologyBuilder();
2
3       builder.setSpout("SpoutOne", new MySpoutOne(), 1).setNumTasks(1);
4       builder.setSpout("SpoutTwo", new MySpoutTwo(), 4).setNumTasks(128);
5
6   builder.setBolt("BoltOne", new MyBolt(), 1).setNumTasks(2)
```

```
        .shuffleGrouping("SpoutOne").shuffleGrouping("SpoutTwo");
```

5.3.4 Walk-through

This subsection will walk through the most essential parts of *SimpleSkylineTopology* described in section 6.5.2 on page 38. Having the explanation of ”Topology example code” from section 5.3.3 starting on page 29 fresh in mind is advantageous. The complete implementation of *SimpleSkylineTopology* can be found in the appendix on page 78. *SimpleSkylineTopology* is illustrated in Figure 16 on page 39. In the figure is spout indicated with S and bolt with B.

”SimpleSkylineTopology.java”

```
19          TopologyBuilder builder = new TopologyBuilder();
20          String topologyName = "SimpleSkylineTopology";
```

Line # 19 creates an instance of *TopologyBuilder* with the most basic topology abstraction level and line # 20 specifies the topology name.

```
26          builder.setSpout("Generator",new BinFileReadSpout(), 1);
```

Line # 26 declares a *BinFileReadSpout* spout in the topology. It is given component ID ”Generator”. The spout is set to have one executor and no task parallelism is indicated, making it one task.

```
28          builder.setBolt("Simple-SL", new SimpleSkylineBolt() , 1).
                localOrShuffleGrouping("Generator");
```

Line # 28 declares a SimpleSkylineBolt in the topology. It is given component ID ”Simple-SL”. It is set to have same parallelism as the spout above. Line # 28 also specify that declared bolt should subscribe to component with the ID ”Generator” using the local or shuffled grouping. The grouping does not matter in this case, as it is only one task to receive the stream.

```
32          Config conf = new Config();
```

```
42              conf.put(CustomConfig.PointDimensions, 5);
45              conf.put(CustomConfig.GeneratorType, DistributionType.
                    ANTICORRELATED.getId());
48              conf.put(CustomConfig.ToggleSleepValue, 2);
51              conf.put(CustomConfig.ToggleSleepTime, 20);
55              conf.put(Config.TOPOLOGY_ACKER_EXECUTORS, 0);
```

Line # 32 creates a configuration object. This object holds topology configuration to be submit together with topology to cluster. The configuration options in line # 42,45,48,51 and 55 is described in section 5.3.1 on page 27 and page 28. The listed settings specifies that it is five dimensions (line # 42), the dataset used should be anti-correlated (line # 45), spout should sleep for 20ms every 2nd time it executes (line # 48 and 51). Line # 55 specifies that it should not be an acker task, thereby turns message reliability off. All tuples will be instantly acknowledged when leaving the spout.

```
102             if (args != null && args.length == 0) {
106                     StormSubmitter.submitTopology(topologyName, conf,
                            builder.createTopology());
116             } else if(args.length > 0) {
```

The if statement in line # 102 is not needed. It was introduced to allow fast switching between submitting defined topology to cluster and running in local debug mode. Line # 106 submits the topology to the cluster, attaching topology name and configuration.

It was decided not to dive into *BinFileReadSpout* and *SimpleSkylineBolt* step by step to avoid this section becoming to extensive. *BinFileReadSpout* and *SimpleSkylineBolt* are fairly well documented in their implementations.

6 Method

Different topologies have been implemented to show differences in tested setups. For every iteration the complexity of those topologies will increase with appliance of techniques described later. The abstraction level used was the most basic one, *TopologyBuilder*. The decision on topology abstraction was done after weighing multiple facts.

I decided not to use Storms ”at least processed once guarantee”, or ”only once guarantee”, because the problem already implies omitting of data from the input stream. The completeness of the algorithm output would not be considerably worse when messages are lost during processing, than omitted on arrival as a part of the problem limitation.

There is another reason for turning down the usage of "only once guarantee". In a worst case scenario where a point would be processed multiple times, the point would either be rejected multiple times or stay as multiple points in the skyline. In a real setting one of the points dimensions would be an identifier enabling the point to rejoin with its data abstracted away on an initial horizontal data-split. The identifier would not be used in skyline calculations, but is used to guarantee uniqueness of final points, and thereby also rejects a point that already is present in the skyline.

The first argument weighs against higher topology abstraction or stronger message guarantee. It is not needed and it comes with unnecessary processing and message overhead. The second argument: There is no need for transactional behaviour, it does not really add anything in this skyline setting. On the other hand, using at least once guarantee would make it possible to control the workers send and receive buffers in a better way as the spout would simply stop emitting new tuples when it hits a predefined limit of unprocessed tuples. The best argument against is; Utilization wise it is best when topology is in a balance where bolts can keep up with spouts. In other cases data will be waiting in full buffers. However Storm allows for specifying internal queues' size and how many tuples a spout can have pending.

The way Storm implement its internal guaranteed message passing is selectively inexpensive. Which alone weigh for using a higher abstraction level as Trident. The Trident abstraction level was considered, but Trident did not support bolts to have multiple output streams. This was the final deal breaker for a higher abstraction level. The issue is described more in depth in [30]. It would be possible to modify Storm source code to the wanted behaviour, but it was not considered the target of this thesis. The problem with only one output stream is demonstrated in the illustrated BroadcastSkylineTopology. (section 6.5.6 on page 41) With Trident, the topology would be impossible. In fact impossible is a bit wrong, as the bolts can funnel all outgoing tuples in the same stream, forcing the same bolts to subscribe to the output from itself, and the others, and then do demultiplexing on all incoming. Anyhow that is not a solution as the bolts would at certain times, output the whole partial skyline in addition to filter points, implying a no doubt overflow death of the topology.

6.1 Constrained Skyline

The experiments have been run with no constraints on any dimension. That has been done with keeping every dimension as the data type Double, but noticeable is the range for each dimension forced between zero and one million.

6.2 Optimizations

The nested loop algorithm was used to compare against every point in the skyline at the arrival. This stands in a great contrast to GridSky. The main reason it differs from GridSky lays on the decision of not using a continuous window. GridSky did optimizations on windows, like removing points that would never be a part of the skyline, in my case a tumbling landmark window was used, as it hopefully should suit the problem better as it is able to consume or process way more tuples to extract information.

A quick recap of the continuous window, typically keep a given number of items, and then add and remove x-points to the window, and do a recalculation over the items in the window. Such an approach imply rerunning over bigger partitions of the data multiple times, and that was considered too expensive.

The optimization done on the tumbled landmark window, was to drop tuples when they were not part of the skyline, as it would never be part of the skyline later either. The tumbling of the window was done by restarting the topology. A better way doing tumbling would be to implement a flush to bolts holding state information like filters and partial or final skylines. TickTuples could be used to initiate such flushes.

Filtering on a value constraint can easily be added to any bolt. It is advantageous that as little as possible of uninteresting points reaches resource demanding bolts. Bolts with a low level of parallelism and a high part of sequential code. Filtering on constraints should be applied early in any topology. The needed logic could be added to a generic *FilterForwardBolt*, all it would need is to instantly reject points when a dimension is not matching a given constrain. More about FilterForwardBolt can be found in section 6.4.3 on page 37.

If datasets were generated to be more similar to real stock data, they would probably have 200 dimensions maybe even more. It would be ideal to sort out points that could be of interest with a filter. A point that is found to be of interest can then be optimized with vertical split before it is sent to the next layer of processing. The reason for doing vertical split is to get the dimensionality of real data points from a stock exchange down to a reasonable number and thereby also size.

Imagine a horrible large skyline with 200 dimensions and only five of them are used to calculate the skyline. In that case the memory foot print of the skyline would be 40 times bigger, dramatically increase the chances for or the number of cache misses. A cache miss is when the Central Processing Unit (CPU) has to go to the next level of cache or in worst case off chip to the RAM to get the needed data. Unnecessary data dimensions elongating cache will waste cache lines obtained for other processes.

6.3 Data generation

The source for generating data with Java, directly to a file, was supplied from my first supervisor at NTNU. The algorithm and most of the code for generating uniform, correlated and anti-correlated was kept as it was. The biggest changes were to adopt it to suit inside a Storm spout.

The reason for adapting it to fit in a Storm spout was, that I could not see any reason for not increasing parallelism. When generating hundreds of millions of numbers, the different generated distribution will not have their correlation patterns changed with a slightly increase of parallelism using different seeds.

For a understanding of the data generation and its topology, please see Figure 14 on page 36. The Java implementation of the topology can be found in the supplied jar archive in the *source* folder under package *no.stud* and *GenerateDatasetTopology.java*. The implementations of the spout and bolt used for generating can be found in their representative packages, under the file-names *BinFileDumpBolt.java* and *GenerateDataset.java* For the generating of data was Storms message passing reliability redundancies not considered needed, or a likely pitfall when left out.

The data generator wrapped in a spout could be used directly as input to the skyline bolts, but the generation of points are an expensive process that would possible drain a little cluster for computing power needed for computing of skylines. For same reason was it decided to use binary data written directly to and from files to feed topologies.

For the actual data generator, configuration option *DimensionMaxValue* was kept at one million for all generated datasets. The number of dimensions was set to 5 and 10 and ran with the Uniform, Anti-Correlated and Correlated generators. For every dataset a minimum of 10GB was generated on every node for later use. The parallelism in term of executors was set to 8 for the spout and 4 for the bolt.

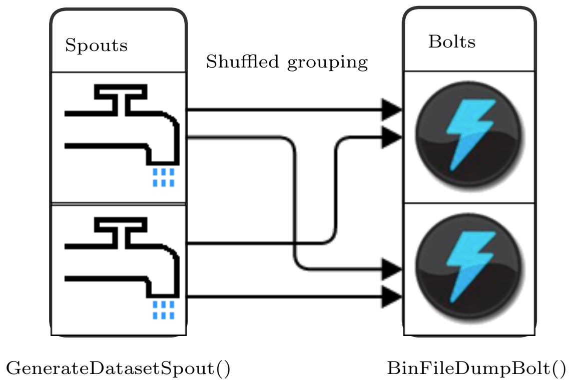


Figure 14: Generate-Dataset-Topology.

6.4 Basic Components

This subsection presents a component breakdown of the most used spouts and bolts.

General info:

- The Java implementation of spouts can be found in the supplied jar archive in the *source* folder under package *no.stud.spouts.*
- The Java implementation of bolts can be found in the supplied jar archive in the *source* folder under package *no.stud.bolts.*

6.4.1 BinFileReadSpout

The *BinFileReadSpout* is the main data source in the represented topologies. As the name indicates, the spout does read a binary file and feeds the content onto a stream. In the cases where the spout had parallelism higher than one was the spout programmed to read different input files using modulation on the task index identification. It was done to avoid feeding the same input multiple times in parallel into the topologies. When it was not done the skyline points would have n copies of itself in skylines, where n represent the number of spout tasks reading the same file. That would lead to a significant lower throughput in topologies, as performance of skyline bolts is closely related to number of skyline points kept in bolts local state.

The *BinFileReadSpout* needs to be configured with how often it should sleep, how long it should sleep in milliseconds, what dataset it should read (in terms of dimensions and correlation type). Detailed information about the configuration options and how to set them can be found in Section 5.3.1 on page 28.

6.4.2 SimpleSkylineBolt

The *SimpleSkylineBolt* is as the name indicates, a bolt that creates a skyline using the nested loop approach. The bolt is making use of TickTuples, and every ten second it will report the current skyline size to the assigned workers standard output. The standard output is written to the workers log on the node, and can be manually collected from there.

6.4.3 FilterForwardSkylineBolt

The *FilterForwardSkylineBolt* is as the name indicates, a bolt that does filtering and forwarding points that pass filter. The points not passing this filter will be rejected and omitted from future processing. A filter can be used to improve overall performance. A filter would typically be used to discard points that do not apply to a given constraint on one or more of the dimensions.

In this case *FilterForwardSkylineBolt* is implemented to do simple pre-skyline consisting of max 100 points, overwriting the oldest point if the pre-skyline is already at the max size of 100 points. To always overwrite the oldest skyline point was a design choice to make sure that the filter would adapt to possible drift in the input stream.

6.4.4 IncrForwardSkylineBolt

IncrForwardSkylineBolt is a implementation based on *SimpleSkylineBolt*. The bolt mainly differs with an output stream. The bolt will emit all tuples that are added to the local skyline onto that output stream. When the bolt emits such a stream, it does allow for the possibility to merge multiple skylines created with different instances of the same bolt to be merged into a final, or more complete skyline. The bolt has no limitation on the size of its local skyline.

6.5 Topologies

This subsection presents a breakdown of certain topologies.

General info:

- The Java implementation of the topologies can be found in the supplied jar archive in the *source* folder under package *no.stud*.
- The stream groupings shown in the figures are covered in section 5.3 on page 25

6.5.1 BlackHoleTopology

The *BlackHoleTopology* is as the name indicates, a topology where bolts act as a black hole. The bolts in the topology use ”local or shuffled grouping” consuming the ”Generator” stream. ”Generator” stream will therefore stay on the node when a *BlackHoleBolt* exist on same node. When the streams stay on same node, it will make use of internal messaging. The experiment with *BlackHoleTopology* was created to gain an really rough indication of how much tuples the compute nodes could manage to process with different message costs.

Test one and two forces all logic to one process, with multiple threads, enforcing usage of LMAX Disruptor for message passing. Test two differs from test one as it will show how sleep value affects the throughput. The third test shows how the throughput in topology will be affected when a node runs multiple workers(processes) instead of multiple executors(threads). In contrast to test one and two is test three making use of ØMQ for the message passing. Test four will show the total maximum throughput when not making use ØMQ as every node will have their own instance of *BlackHoleBolt*.

The *BlackHoleTopology* is implemented in *BlackHoleTopology.java* and Figure 15 (on page 39) shows how spouts and bolts are utilized and connected, in the implementations.

6.5.2 SimpleSkylineTopology

This is the basic implementation of a nested loop skyline in a stream environment. It is simple in the way that we can have one or multiple spouts feeding only one *SimpleSkylineBolt*. The *SimpleSkylineBolt* will consume points from its input buffer and check them one by one, at arrival against already existing skyline in the bolt. The bolt is not capable of having parallelism higher than one, as it is nothing in this topology to combine the skyline points from different partial skyline sources. In fact the bolt has no output stream.

For inputs that generate big skylines, this topologys *SimpleSkylineBolt* most

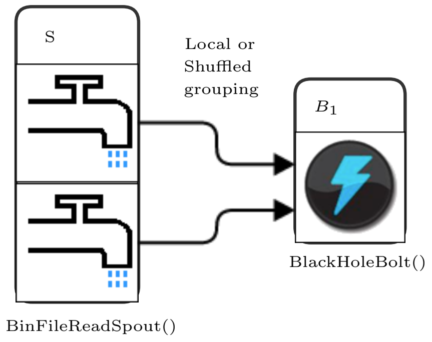


Figure 15: Black-Hole-Topology.

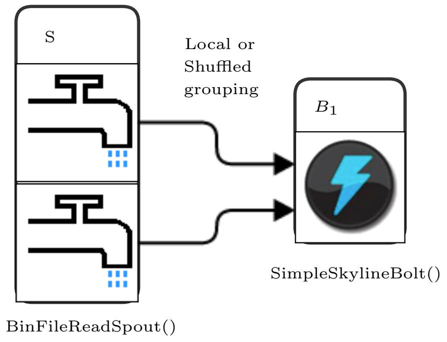


Figure 16: Simple-Skyline-Topology.

likely will be CPU and later on memory bound. Bolt speed is doomed to decay over time when skyline size m increase. That will possible lead the heartbeats to starvation, and the bolt will most likely die or be killed during a forced restarted from the supervisor, that believes it is dead.

The *SimpleSkylineTopology* is implemented in *SimpleSkylineTopology.java. SimpleSkylineTopology.java* is explained in section 5.3.4 on page 31, and is thereby also represented in the appendix on page 78 . Figure 16 on page 39 shows how spouts and bolts are utilized and connected, in the implementations.

6.5.3 FilteredSimpleSkylineTopology

The topology has its name from the fact that it is the *SimpleSkylineTopology* combined with one executor running two *FilterForwardBolt* tasks. The thought behind

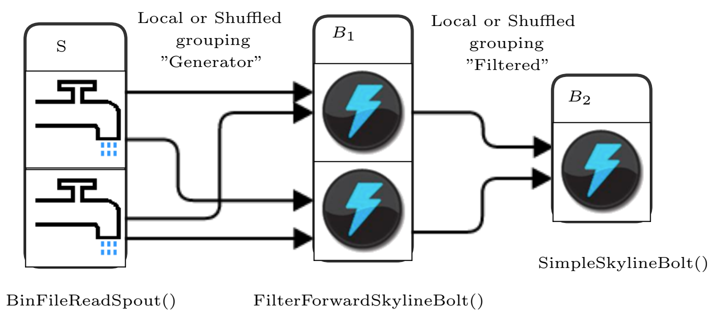

Local or Shuffled grouping "Generator"
Local or Shuffled grouping "Filtered"
S
B_1
B_2
BinFileReadSpout()
FilterForwardSkylineBolt()
SimpleSkylineBolt()

Figure 17: Filtered-Simple-Skyline-Topology.

this test, was how a simple filter would impact the performance of the original *SimpleSkylineTopology*. This topology still has the limitations of the *SimpleSkylineTopology* when the filter has a low filter rate.

The *FilteredSimpleSkylineTopology* is implemented in *FilteredSimpleSkylineTopology.java* and Figure 17 (on page 40) shows how spouts and bolts are utilized and connected, in the implementations.

6.5.4 ImprovedSkylineTopology

The *ImprovedSkylineTopology* is an improvement over the *SimpleSkylineTopology* allowing parallelism like the one seen in BOCS monitoring architecture (see Figure 4 on p. 10.). Having a look at the topology representation in Figure 18 on p. 41. The *IncrForwardSkylineBolt* is making a partial skyline of its input stream and is emitting every increase to the final skyline that is made with the *SimpleSkylineBolt*.

The *ImprovedSkylineTopology* is implemented in *ImprovedSkylineTopology.java* and Figure 18 (on page 41) shows how spouts and bolts are utilized and connected, in the implementations.

6.5.5 FilteredImprovedSkylineTopology

The *FilteredImprovedSkylineTopology* builds on the same design as the *ImprovedSkylineTopology*. It mainly differs because it adds a filter layer (*FilterForwardSkylineBolt*) in front of the *IncrForwardSkylineBolt*.

The filtered improved skyline topology is an extension of the improved skyline topology. The main difference here is that the topology is extended with additional

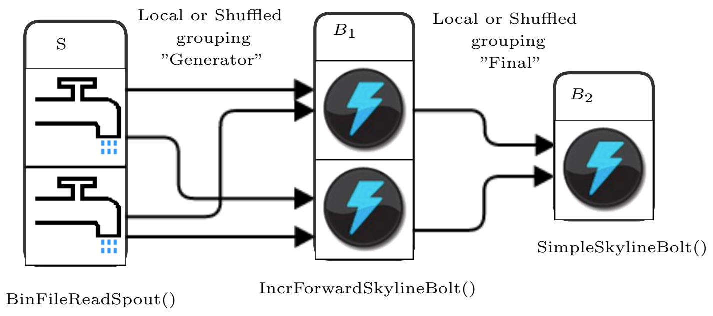

S
Local or Shuffled grouping "Generator"
B_1
Local or Shuffled grouping "Final"
B_2
SimpleSkylineBolt()
BinFileReadSpout()
IncrForwardSkylineBolt()

Figure 18: Improved-Skyline-Topology.

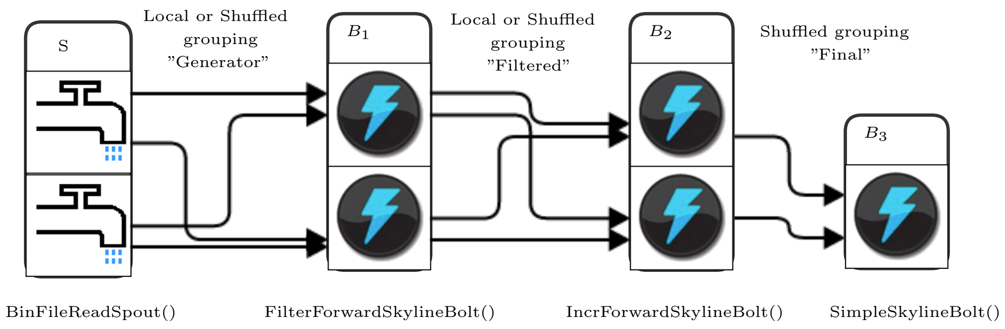

S
Local or Shuffled grouping "Generator"
B_1
Local or Shuffled grouping "Filtered"
B_2
Shuffled grouping "Final"
B_3
BinFileReadSpout()
FilterForwardSkylineBolt()
IncrForwardSkylineBolt()
SimpleSkylineBolt()

Figure 19: Filtered-Improved-Skyline-Topology.

layer of bolts to do filtering. That will cause some extra delay on the path before the points makes it into the final skyline.

The *FilteredImprovedSkylineTopology* is implemented in *FilteredImprovedSkylineTopology.java* and Figure 19 (on page 41) shows how spouts and bolts are utilized and connected, in the implementations.

6.5.6 BroadcastSkylineTopology

The BroadcastSkylineTopology is allowing parallelism. *BroadcastSkylineBolt* holds a portion of the skyline in its memory. Whenever it is possible, the bolt should not hold on to a bigger skyline than what it can fit into the CPU's level two cache. *BroadcastSkylineBolt* processes tuples and broadcast when it adds a point to the skyline. Unless points are lost, added points will be processed off all *BroadcastSkylineBolt*. *BroadcastSkylineBolt* will use the received broadcast as filter-points and

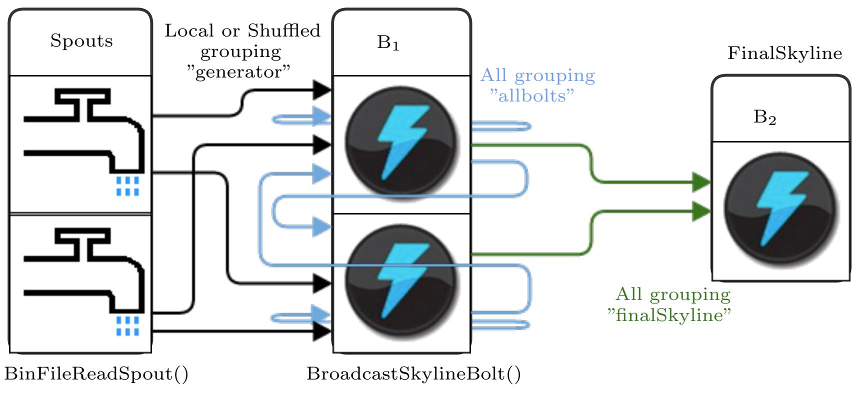


Figure 20: Broadcast-Skyline-Topology.

therefore will not keeping them.

Overall processing cost in the BroadcastSkylineTopology will be higher compared to the other skyline topologies. That because it requires a lot more communication and overhead in processing, as every skyline point will be broadcasted. Therefore, all the skyline points will be processed by every partial skyline.

Another disadvantage is when stream X seems to always be worse than stream Y. In that case the BroadcastSkylineBolt reciving stream X may add a lot of false positives to its local skyline and spam bolts intercommunication stream. Therefore a better approach would be to have filter in front that receives filter points from a broadcast stream of filter points. *BroadcastSkylineTopology* have a final bolt that assemble the final skyline on a time based interval defined with eg. Tick tuples. Partial skylines would needed to be merged, as it is no guarantee that the messages and partial skylines will be in sync when submitted. The topology is represented in Figure 20 on page 42.

7 Results and Discussion

7.1 Results

Results will be presented with the number of tuples processed during a 10 minute period, measured from start-up of topologies. Tests run with topologies using only one worker, were run sequentially on the same *i5* node. That was done to eliminate the difference from slightly different hardware in the cluster. Unless otherwise is specified; *taskParallelism* of spouts and bolts is one to one with number of executors assigned. A quick recap, *Executor* is equal to a thread.

General table explaination:

- # is the line identifier in the table.
- **DD** is short for Data Distribution, and indicates if the test data run with is of (U)niform, (C)orrelated or the AntiCorrelated(AC) type.
- **Nodes** indicates how many nodes that are utilized.
- **Executors** indicates how many Executors that are utilized.
- **Workers** indicates how many Workers that are utilized.
- **Msg/s** is short for messages per second. Indicating how much input data the topology is consuming from the spouts. Calculated over a ten minute interval. (The ten min total consumed divided by 600.)
- **Capacity** indicates the load on a bolt or group of bolts. Calculated over a ten minute interval.
- **Execution** indicates the average time spent in logic of a bolt, or group of bolts over the last ten minutes.
- **Processed** indicates the total number of messages in the stream, consumed by a bolt. In topologies with one bolt will that be the same as total of spout produced.
- **Skyline** indicates the size of the Skyline at ten minutes, measured with tick tuples.
- **ms** indicates the sleep time in ms.
- **S** is the number of spouts in topology.
- **N/A** indicates that the measure is not applicable.
- **Fail** indicates that a topology failed partially or totally within ten minutes.
- **B**$_n$ is short for the parallelism hint for that Bolt, where $_n$ indicates which bolt it is, numbered from left to right in the topology. So the first bolt after a spout would be $n = 1$ the second $n = 2$ and continues.
- **PB**$_n$ is short for Processed Bolt, where $_n$ indicates which bolt, numbered from left to right in the topology. So the first bolt after a spout would be $n = 1$ the second $n = 2$ and continues. Processed is explained above.
- **CB**$_n$ is short for Capacity Bolt, where $_n$ indicates which bolt, numbered from left to right in the topology. So the first bolt after a spout would be $n = 1$ the second $n = 2$ and continues. Capacity is explained above.

- **EB**$_n$ is short for Execution Bolt, where $_n$ indicates which bolt, numbered from left to right in the topology. So the first bolt after a spout would be $n = 1$ the second $n = 2$ and continues. Execution is explained above.

7.1.1 BlackHoleTopology

Tests numbered 1 to 3 was performed serially on the *i5* node, so the difference in hardware was eliminated as a source of error. The tests were run for ten minutes with five dimensional data. Since the stream grouping used is ”local or shuffled” and it existed a local bolt, test four utilized the 4 available nodes, showing the maximum number of processed messages on those four nodes not utilizing ØMQ.

The dimensionality of the data sets have an impact, but not big compared to using it in a skyline. The ten dimentional sets were therefore not tested. The tests do close to nothing with the data. All data received by the bolt got dereferenced after some simple instructions and were left for the garbage collector to collect. The results are shown in Table 1.

#	Nodes	Workers	Executors	Processed	msg/s	Sleep	ms	Capacity	Execution	S	B_1
1	1	1	4	20 562 040	34.3k	15	1	0.023	0.001	3	1
2	1	1	4	123 318 800	205.5k	-1	N/A	0.106	0.001	3	1
3	1	4	4	72 279 340	120.5k	-1	N/A	0.072	0.001	3	1
4	4	4	16	485 529 900	809.2k	-1	N/A	0.130	0.001	12	4

Table 1: BlackHoleTopology results.

7.1.2 SimpleSkylineTopology

When running more than one spout per node, the spouts were programmed to make use of different input files to avoid a skyline having 3 equal points for every point in the skyline. Tests were done with one worker, running for ten minutes. Test # 1 to 6 was run sequential for both the ten and five dimensional set on the same *i5* node.

The results from testing this topology can be found in Table 2 on page 45.

(a) One worker, one *SimpleSkylineBolt* and **ten** dimensions.

#	DD	Processed	Msg/s	Skyline	Capacity	Execution	Sleep	ms	S
1	AC	87080	0.2k	88 060	0.429	2.956	2	20	1
2	AC	Heartbeat failure					4	10	1
3	U	Heartbeat failure					5	1	2
4	U	519 680	0.9k	84 408	0.920	1.064	4	4	1
5	C	7 410 520	12.4k	20 082	0.944	0.076	-1	N/A	3
6	C	8 432 400	14.0k	4 641	0. 607	0.043	5	1	3

(b) One worker, one *SimpleSkylineBolt* and **five** dimensions.

#	DD	Processed	Msg/s	Skyline	Capacity	Execution	Sleep	ms	S
1	AC	249 180	0.4k	165 200	0.937	2.384	4	6	1
2	AC	Heartbeat failure					2	1	1
3	U	8 618 940	14.4k	8 126	0.753	0.052	-1	1	3
4	U	15 081 500	25.1k	9 795	0.954	0.038	10	1	3
5	U	11 465 980	19.1k	7 456	0.788	0.041	7	1	3
6	C	41 779 860	69.6k	2 223	0.864	0.012	-1	N/A	3

Table 2: SimpleSkylineTopology results.

7.1.3 FilteredSimpleSkylineTopology

The results from testing this topology can be found in Table 3 on page 45. It is also correct to state that filter bolt had a task parallelism of two, except for test #1 with AC that had one executor and one task. Having the task parallelism of two is making it possible to assign additional executor to the filter bolt. However that was not utilized.

The results from testing this topology can be found in Table 3 on page 45. Comparing the message throughput in this topology with the results from SimpleSkylineTopology shows how the simple filter impacts performance.

One worker,one *FilterForwardSkylineBolt*, one *SimpleSkylineBolt* and **five** dimensions.

#	DD	PB_1	PB_2	Msg/s	Skyline	CB_1	EB_1	CB_2	EB_2	Sleep	ms	S
1	AC	230 480	229 360	0.4k	165 078	0.709	3.274	0.709	2.057	4	6	1
2	U	9 713 760	4 705 860	16.2k	6 868	0.286	0.018	0.555	0.071	10	1	2
3	U	14 192 200	7 463 420	23.7k	8 452	0.286	0.012	0.661	0.053	10	1	3
4	U	16 211 140	11 195 400	27.0k	10 871	0.820	0.030	0.919	0.049	20	1	3
5	C	23 782 860	19 460	39.6k	1 387	0.173	0.004	0.002	0.059	20	1	3
6	C	91 230 380	65 420	152.1k	4 920	0.583	0.004	0.016	0.148	-1	N/A	3

Table 3: FilteredSimpleSkylineTopology results.

7.1.4 ImprovedSkylineTopology

ImprovedSkylineTopology shows message throughput when utilizing of all computing nodes in the cluster. However it is not scaling well for anti-correlated inputs. The reason is very familiar to why the SimpleSkylineTopology is not scaling that well. *SimpleSkylineBolt* tailoring the final skyline, can not keep up with input when cardinality of the skyline set has grown to big.

The results from testing this topology can be found in Table 4 on page 46. When it was less than 4 spouts in this topology, parallelism of *IncrForwardSkylineBolt* was set to number of spouts.

Four workers, four/two *IncrForwardSkylineBolt*, one *SimpleSkylineBolt* and **five** dimensions.

#	DD	PB_1	PB_2	Msg/s	Skyline	CB_1	EB_1	CB_2	EB_2	Sleep	ms	S
1	AC	360 120	70 980	0.6k	17 347	0.324	0.843	0.059	0.502	4	20	2
2	AC	684 340	81 020	1.1k	19 331	0.503	0.668	0.061	0.450	4	10	2
3	U	43 012 900	78 460	71.7k	13 053	0.813	0.041	0.051	0.390	7	1	12
4	C	43 547 920	11 360	72.6k	2 425	0.430	0.014	0.002	0.096	7	1	12
5	C	194 766 580	9 640	324.6k	7 972	0.881	0.008	0.004	0.240	-1	N/A	8
6	C	Failed, ØMQ uncontrolled memory usage.								-1	N/A	9
7	C	Failed, ØMQ uncontrolled memory usage.								-1	N/A	12

Table 4: ImprovedSkylineTopology results.

7.1.5 FilteredImprovedSkylineTopology

FilteredImprovedSkylineTopology shows message throughput when utilizing of all computing nodes in the cluster in combination with filters. It will therefore be natural to compare it with *ImprovedSkylineTopology* that also utilize all the computing nodes without filter. *FilteredImprovedSkylineTopology* is not expected to scale better than *ImprovedSkylineTopology* for anti-correlated inputs. For a topology too scale better with high cardinality on final skyline set, a direct targeting of skyline cardinality would be needed.

The results from testing this topology can be found in Table 5 on page 46.

Table split, test settings.

Four workers, four/two *FilterForwardSkylineBolt*, four/two *IncrForwardSkylineBolt*, one *SimpleSkylineBolt* and **five** dimensions.

#	DD	PB_1	PB_2	PB_3	Msg/s	Skyline	CB_1	EB_1	CB_2	EB_2	CB_3	EB_3	Sleep	ms	S
1	AC	563 240	561 780	104 620	0.9k	27 763	0.024	0.037	0.429	0.511	0.235	1.335	4	10	2
2	AC	Failed, ØMQ uncontrolled memory usage.											4	5	2
3	U	67 387 560	41 881 860	63 200	112.3k	16 183	0.839	0.022	0.986	0.048	0.035	0.330	20	1	12
4	C	324 431 260	109 320	12 800	540.7k	8 679	0.555	0.003	0.006	0.104	0.006	0.262	-1	N/A	8
5	C	303 278 640	337 360	22 380	505.5k	11 486	0.523	0.004	0.031	0.132	0.031	0.850	-1	N/A	12

Table 5: FilteredImprovedSkylineTopology results.

7.1.6 BroadcastSkylineTopology

Topologies based on *BroadcastSkylineTopology* was not implemented and therefore naturally not measured. Topologies based on BroadcastSkylineTopologies was work in progress mainly targeting, memory usage thereunder cardinality of partial skylines. However time disposable to finish this thesis put an effective limitation on further research on this topic.

7.2 Discussion

7.2.1 Environment

The Storm cluster was realised of cheap commodity hardware. A typical node consist of a 4 core based Intel i5 or AMD Phenom II in the 2.5-3.5 Ghz range with at least 4 GB RAM available for each worker process. In total Storm cluster consist of four worker nodes running one Storm Supervisor each, with a total of 4 available workers.

One single coordination node contains Zookeeper cluster, Storm UI and one Storm Nimbus instance. Monitoring and fail-safeing of Storm Nimbus, Storm UI and Zookeeper was done with a service named Supervisor. The same service was utilized on the compute nodes to monitor Storm Supervisor.

Nodes have 1 Gb star network connected through a switch except for coordinator node that has 100 Mb connection and slightly weaker hardware. Having slightly weaker hardware and slower network for coordination node was not a bottleneck. Zookeeper is used to hold cluster coordination data, dictated by Nimbus administering the topologies. It was further confirmed with running 'ifstat -S' for network usage and 'top' for processing power. It is a rather small cluster, but it provided great performance and the ability to measure message throughput in Storm topologies. The cluster is shown in Figure 21 on page 48.

Compute nodes Hard Disk Drive (HDD) performance was not considered a problem. For nodes accessing an external HDD over the Universal Serial Bus 2 (USB2), sequential capacity was measured to approximate 31.3MB/sec on single sequential reads. The single read was measured with the Linux command *'sudo hdparm -t /dev/sda'*. *hdparm -t* is doing a raw read, implying that the file system overhead was not included in resulting measurements.

Notable; When more than one *BinFileReadSpout* was running on the same node, reads would not be strictly sequential. However *BinFileReadSput* was implemented using buffered reading and therefore reads bigger chunks for each disk access. For cases with e.g. four *BinFileReadSpout* on the same node, would it be four sequential

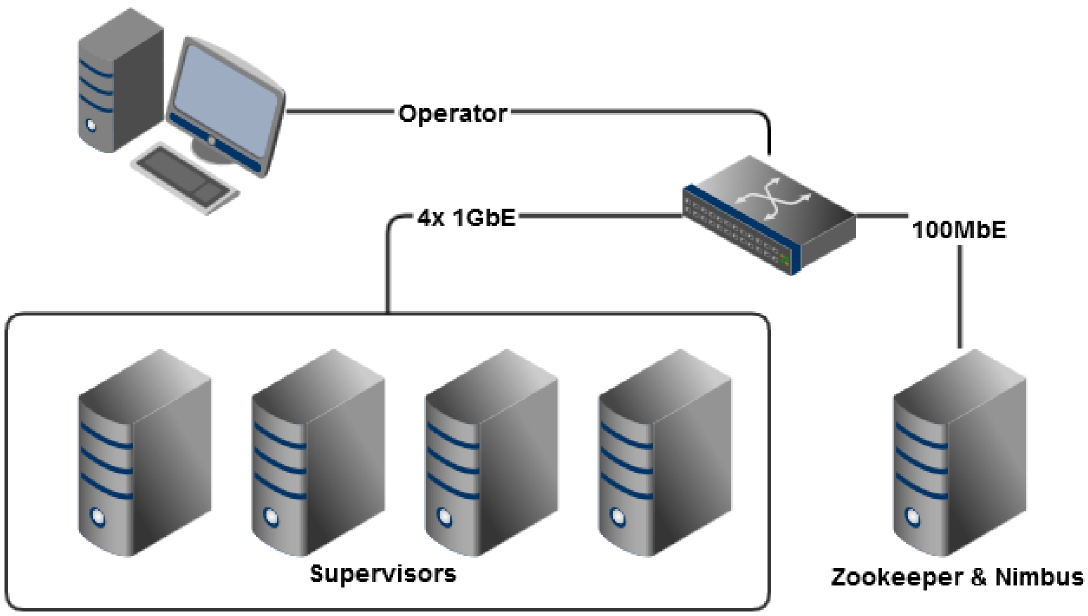


Figure 21: Storm cluster environment.

streams read from HDD. Files on disk was written sequentially one at the time during the generation. In the case of three readers on the same node was the reads from the HDD spotted to be approximately 5.87MB/s, the utility used was 'iotop'.

It was also done measurements of the network. It was done using the Linux command *'iperf'*. *iperf* is a tool made for doing network throughput tests. Two of the compute nodes was set up as *iperf* servers and having another compute node connecting to it. The command used to connect compute nodes was *'iperf -c 10.0.0.X -d -t 60 -i 10'*. the arguments is; c - client, d - bidirectional test, t - how long the test should run, i - on what interval the throughput should be reported. Two such tests were run in parallel with the bidirectional streams, implying four streams. Resulting throughput for those four streams was better than 850 Mbit-s/sec on all streams for all report intervals, for both directions. With every compute node capable of having over 100 MByte/sec in and out, it was concluded that the network was no bottleneck. However it needs to be mentioned that no inspection on how Storms TCP packet size would be the same as used in the test. The test utilized the operative systems standard value.

7.2.2 Datasets

A synthetic dataset was used as it allows extremely many values in each dimension, and thereby allowed to pressure the limits of what the topologies could handle. Compared to a real dataset from a stock exchange where dimensions often will be more limited in their ranges, and indirectly reduce number of possible skyline points m in a possible endless stream.

I decided on sticking with 5 and 10 dimensions for the generated correlated, anti-correlated and uniform datasets. Mainly to be able to run tests with different computing loads. Doubling of the dimensions had a huge impact on the performance in several ways. It allows a much faster increase in the size of the total skyline. Ten dimensions instead of five also double numbers of elements that are compared when comparing two points. The cost of generating the static datasets was not measured as it is considered not relevant to the skylines performance.

From the results of the Simple-Skyline-Topology in Table 2 can we see that the topology had issues not breaking down even with one spout feeding the topology. That was especially for the AntiCorrelated and Uniform distributions, result line 1-4. It was very time consuming to find a balance for, how often the spout should sleep. For how long it should sleep, and how many of them it should be. To not spend to much time on tweaking, was it decided to drop future testing of the ten dimensional set and focus on the five dimensional one.

With point generator wrapped inside a spout, generating spouts could serve tuples directly as input to bolts. That was not utilized. Generating points are an expensive process of generating randomness, that would possible drain a little cluster for a lot computing power. In this case; Computing power needed for computing partial and final skylines. For the same reason of not utilizing direct generation, it was decided to use binary data written directly to and from files to feed topologies.

7.2.3 Weaknesses

In the proposed methods for skyline topologies, a point is considered valid until the window is tumbled. That is chosen by design for being able to handle a massive flood of data. However it can be argued for that a point(an imaged stock interests), only will live for a short period of time. That fact can not be denied, but it can be given an account for. In the tumbled landmark window it would be ideal to filter short living illegitimate stock interests at arrival of points, when discovered. That can be done using filters. However, a point that is considered invalid for other reasons can be removed at any time, without hurting the completeness of the algorithm used. This decision of removing or keeping invalid points until the window is tumbled would be left to the stock trader.

Changing to a sliding window would possible keep better track of validity of points. However it will imply a very small window size compared to tumbled landmark. Sliding window could be optimized with graphs or networks calculating skyline candidates and eliminate no candidates. But those optimizations do not change the fact that a sliding window can process way less as it implies some extra processing and storing of possible candidates. When a point is taken out from the skyline, the complete skyline is recalculated unless it is known which points the removed point dominated. And still, it might be other points dominating the same points. Keeping track of candidate points is complex and will most likely lean toward a micro batch system for exploiting data locality better during reprocessing.

If ageing and decaying of points was utilized for a landmark window, it would still give a guarantee that returned points will be among the best, but necessarily not the best set from an operators point of view. A point matching a stock trader's current choice, might decay and the closest stock offer is not represented in the skyline as the decayed point just dominated it. To give a better guarantee it is possible to design implementations so points dominated by other points that are about to decay would be kept for a short while. However, this again will lead to a bigger memory footprint, higher processing cost, and less available space in the level two cache. A suggestion would be to stick with a tumbling window, and look into partial tumbling as an operator might prefer not starting from scratch every ten minutes.

Validity of points is considered a weakness, it is very complex to extract a stock trader's validity policy without closer co-operation with a stock trader. A stock trader's point validity policy might vary with direction of stocks main index. Validity policy could undergo more work to verify right design choice.

Generated datasets could have their linear coefficient correlation values calculated, and confirm correct distributions. If slightly erroneous correlations exist it would not effect this thesis' prototyping purpose. However, skyline cardinality is strongly connected to dataset correlation. For performance the important part will remain skyline cardinality, as it effects the time we need to process one input tuple.

CPU frequencies on compute nodes was set to regulate them-selves according to CPU load. It is the default setting for Ubuntu server. This undesirable configuration was discovered after all measurements were completed, hence no way back to set all the CPU core frequencies to a fixed value.

Testing of topologies was not done multiple times and an average was not calculated, because of this thesis' time constraints. Testing therefore deviate from established practices. It also needs to be highlighted that results in Table 2 and

3 might be slightly erroneous. After testing and interpreting an error in the code was discovered. It could possible affect which order input files were utilized, when spouts parallelism was not maximised. Correct distribution and dimensions were always used, but utilizing of sets in the same order was therefore not guaranteed and might have effected the results slightly. When inspecting the results in Table 5, when comparing results obtained in row # 5 and row # 6, something does seem a bit odd. It should not be affected of the programming error mentioned above, however, those numbers speak for themselves and clearly indicates something is wrong. Suspecting multiple points of the same values exist, issue might be from a human error related to a recovery of datasets after a fatal HDD crash. Time for further investigation was not available.

7.2.4 Performance

When utilizing one node a great performance boost was discovered when comparing no filter versus filter usage. Comparing FilteredSimpleSkylineTopology results (Table 3 on page 45) with the SimpleSkylineTopology results (Table 2 (b) on page 45). Particularly for row # 6, correlated distribution, it is clearly visible that for the correct type of data distribution even the simple filter is very efficient. This is also confirmed by comparing the results of ImprovedSkylineTopology(Table 4 on page 46) with FilteredImprovedSkylineTopology(Table 5 on page 46).

Comparing row # 6 in FilteredSimpleSkylineTopology featuring 91.2 million messages processed versus row # 6 in SimpleSkylineTopology's 41.8 millions. FilteredSimpleSkylineTopology has a performance improvement of 118% over the SimpleSkylineTopolgy for the correlated dataset. For anti-correlated and uniform was no significant difference observed. Comparing PB_1 versus PB_2 in FilteredSimpleSkylineTopology, it is clear that filtering has a big impact for uniform data. However, the additional cost of extra context switches and processing, seems not to be able to payback the extra cost without having additional CPU cores to exploit. Filter is observed to have close to no significance when processing anti-correlated streams.

A Performance improvement of 66.5 % was observed utilizing filter for ImprovedSkylineTopology, comparing row #5 in ImprovedSkylineTopology with row # 4 in FilteredImprovedSkylineTopology. Comparing PB_1 versus PB_2 for FilteredImprovedSkylineTopology reveal the same as mentioned in the paragraph above, for uniform and anti-correlated.

The performance boost of utilizing multiple nodes without effect of filter is shown when comparing SimpleSkylineTopology with ImprovedSkylineTopology. Here again is uniform and particularly anti-correlated scaling worse. It is also possible to compare with BlackHoleTopology that indicates practical maximum for different settings.

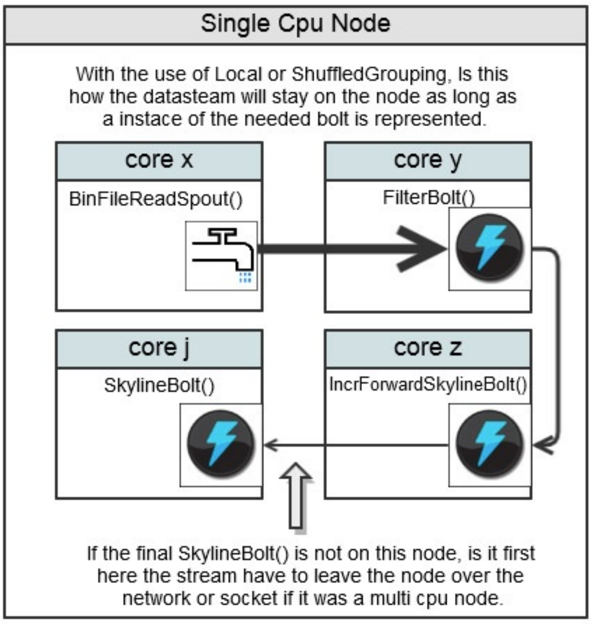

Single Cpu Node
With the use of Local or ShuffledGrouping, Is this how the datasteam will stay on the node as long as a instace of the needed bolt is represented.
core x
BinFileReadSpout()
core y
FilterBolt()
core j
SkylineBolt()
core z
IncrForwardSkylineBolt()
If the final SkylineBolt() is not on this node, is it first here the stream have to leave the node over the network or socket if it was a multi cpu node.

Figure 22: Possible utilization on single CPU node.

Number of workers on one node effects Storms performance significantly. It might be tempting to run multiple workers on a single multi-core CPU. That should be avoided unless the workers are supposed to run different topologies at the same time on one node. Running multiple workers will lead to a major increased of context switching and a major increase in communication overhead. Going from internal thread to thread communication to communication over the operative-systems socket layer is expensive.

Underlying architecture is very important to understand in more ways, and to play on its premisses to get improved performance. What a close to ideal scheduling of a topology like e.g. FilteredImprovedSkylineTopology would look like is shown in Figure 22 on page 52. Scheduling executors onto nodes using Local or Shuffled Grouping, making it possible for the data-stream to occasionally stay on chip and flow through level two cache.

JVM Garbage Collector is known to impact performance of Java applications. The impact is known to cause freezes of applications running for a short period of time. Because of a problem introduced in next section, the Garbage Collector (GC) was tuned to be very aggressive and it was therefore also consuming way more CPU time than strictly needed. That has of-course coloured the overall performance, but was intentional left on to enforce a difficult circumstance that would occur occasionally with time anyway.

7.2.5 Trouble

During the use of Storm I have run into a lot of different problems. One of the most annoying one as a newcomer was the following experience:

Problems raised when a topology was created and run with the reliability (guaranteed message passing) turned off. The described problems also apply for the case where the Storm *TOPOLOGY_MAX_SPOUT_PENDING* is set to a arbitrary high or unlimited value. *TOPOLOGY_MAX_SPOUT_PENDING* act as a limit for emitting tuples onto the streams. The limit is *task* local. When the limit is hit, the spout will stop emitting new tuples until it has received an acknowledge or failed a tuple.

With no tracking of tuples and spouts emitting onto streams with no limit, given the tight loop described in section 5.1 on page 23, the ack() and fail() functions do nothing, and nextTuple() consume the given CPU timeslot emitting tuples nonestop. (Given that spouts have tuples to emit, something it has in endless streams.) It was suggested to have it sleep for a short amount of time in cases spout had no input to send. That was not the case and a sleep policy was therefore not applied at first.

Carrying on with spouts having no throttling policy. They were always ready for scheduling, and always had work to do. That in combination with a fair CPU scheduler lead to spouts flooding streams. Bolts had heavier logic and could not stand a chance to keep up with spouts. But still, it should not be that dramatical with full internal buffers? With no reliability turned on, it would still be normal to expect some kind of flow control, either back pressure or overflows. In the case of back pressure would a slowdown of a component in topology cause slowdown of component in front of that component. In the case of a overflow tactic it would be to discard messages when buffer is full.

Regardless of utilized flow control, one node running a worker with such a workhorse of a spout described above, seemed to run out of memory insanely fast. Its CPU usage was also noticeable close to maximum. The first thought was that something in the code was not dereferencing proper, but nothing supporting that hypothesis was found. Could it simply be that the JVM GC could not keep up with cardinality of points created and discarded? JVM GC was configured to be aggressive and allowed to spend a lot of time. However, the fault still persisted, enforcing a thorough dive. Starting with inspecting Java heaps where nothing particular was discovered. Moving on inspecting the JVM Storm Worker process it slowly came to light that ØMQ was causing the memory consumption, with its internal socket buffering.

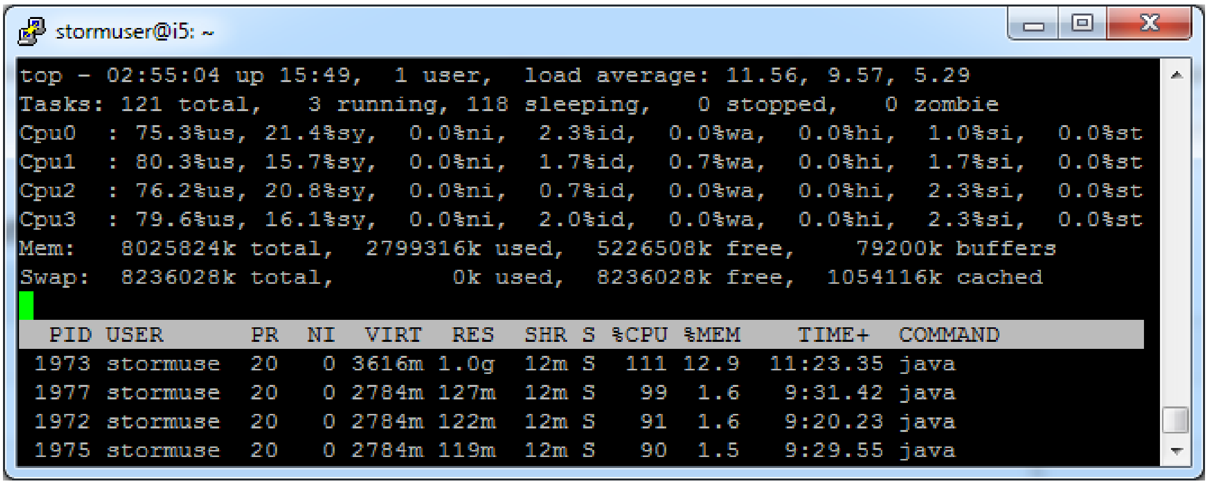
With the memory consumption localized. Possibilities to elevate it were tried. One possibility of setting the "High water mark" on ØMQ sockets making them block when specified buffer size is reached, was utilized hoping for a back pressure effect. It sadly showed quickly to be causing starvation of heartbeats. Realizing that underlying ØMQ worker to worker communication could not utilize back pressure or discard overflowed packages the step towards a throttling policy for spouts was done. The alternative that includes Storms *TOPOLOGY_MAX_SPOUT_PENDING* configuration option is not available without tuple tracking.

One pattern was discovered after running a lot of tests, making it possible to recognizance when ØMQ was about to cause a major fault. The common pattern was discovered to be when a bolt or a group of bolts have acked a lot more or less than it should from a stream. With reliability turned off every received tuple will be instant acknowledged and counted for statistics. To confirm the fault when pattern is showing, a Java utility like 'jvmtop' combined with the usage of 'top' would be handy. 'jvmtop' will show a breakdown of the Java process' heaps. Those heaps would typically show to be of expected size, but when the same JVM process is inspected in top will it quickly reveal that JVM worker process is heading towards consuming all available RAM.

For a reproduced situation in a controlled manner; a simple node was set to run four workers to force traffic over ØMQ sockets using the BlackHoleTopology. Have a look at Figure 23 on page 55, to see how memory usage and processing is slowly running out of control.

Another annoying issue is when the topologies are partially crashing, it is often a result of high latency on a components execution in combination with a JVM Garbage Collector (GC) run.

I also ran into a lot of other different sized problems and error messages. Some errors and their plausible solutions are to be found in the appendix section 9.6 on page 74.



```
top - 02:55:04 up 15:49,  1 user,  load average: 11.56, 9.57, 5.29
Tasks: 121 total,   3 running, 118 sleeping,   0 stopped,   0 zombie
Cpu0  : 75.3%us, 21.4%sy,  0.0%ni,  2.3%id,  0.0%wa,  0.0%hi,  1.0%si,  0.0%st
Cpu1  : 80.3%us, 15.7%sy,  0.0%ni,  1.7%id,  0.7%wa,  0.0%hi,  1.7%si,  0.0%st
Cpu2  : 76.2%us, 20.8%sy,  0.0%ni,  0.7%id,  0.0%wa,  0.0%hi,  2.3%si,  0.0%st
Cpu3  : 79.6%us, 16.1%sy,  0.0%ni,  2.0%id,  0.0%wa,  0.0%hi,  2.3%si,  0.0%st
Mem:   8025824k total,  2799316k used,  5226508k free,    79200k buffers
Swap:  8236028k total,        0k used,  8236028k free,  1054116k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 1973 stormuse  20   0 3616m 1.0g  12m S  111 12.9  11:23.35 java
 1977 stormuse  20   0 2784m 127m  12m S   99  1.6   9:31.42 java
 1972 stormuse  20   0 2784m 122m  12m S   91  1.6   9:20.23 java
 1975 stormuse  20   0 2784m 119m  12m S   90  1.5   9:29.55 java
```

Figure 23: Failing ØMQ worker to worker communication.

8 Conclusion and Recommendations

8.1 Conclusion

The main goal of this thesis was to introduce a low latency skyline computation in a stream environment. This goal was reached. Skylines was computed ”on the fly” instead of ”on request”. The different skyline computations were implemented on top of Storm utilizing different strategies to exploit parallelism and optimization filter. The overall performance with Storm appeared to be stunning, specially when accounting for running on mid-end commodity hardware.

Processing thousands of messages per second was a pleasure with Storm. To use the most basic abstraction level, was a medium success. It enforced a whole lot of work with the basic concepts of the underlying architecture and inner workings of Storm. That to make sure Storm and the frameworks Storm relies on did behave as expected. With underlying issues sorted out, and with fairly large skylines around and above ten thousand points; The small cluster did manage to process above 100 thousand messages per second for uniform distribution and above 500 thousand for correlated distribution, both with five dimensional points. However, processing anti-correlated distributed datasets was showing a disappointing performance. Those datasets lead to high cardinality on skylines sets with an accompanying high processing cost. Bottlenecks observed are sequential skyline merge processes. Processing cost is increasing significantly for bolts when cardinality of skyline set outgrow CPU level two cache. For anti-correlated datasets that happens rapidly, and leads to the observation of a disheartening performance. Topologies using other strategies for utilizing more processing power will need further research. Bolts collaborating on holding a skyline in their level two cache could be one way to go.

The introduction of filter showed small filters containing up to hundred points, could more than significantly reduce the number of points that need to be forwarded for future processing with correlated distribution. Same filter showed a significant reduction for uniform distributions. For anti-correlated distributions could a small reduction be observed, probably not worth the filter processing cost.

Personally an important discovery was; Always when a part of the process related to this thesis was done, I moved on to the next stage. It then magically showed how the previous part should have been iterated over again, as improvements would be possible by simply doing something slightly different. However, after X-iterations over different parts of the thesis with accompanying improvements, I was forced to move on because of the time constraint. It was very visible for the part containing time consuming testing. It was simply no way back. It was a great experience. However, I personally wished to reach working implementations earlier, so some time could be allocated to target higher cardinality skylines on top of Storm.

8.2 Recommendations

This thesis covers some possible implementations of the skyline operator over streams with Storm framework. However it is still much more that can have further attention in this specific research path.

- Targeting high cardinality skylines.

Other obvious improvements and extensions that are directly related and might be introduced to my existing work are:

- Utilizing smaller batches through topologies, it is not really realistic to pull a single tuple from the stock exchange source at the time.
- Test with more realistic data, and utilize both vertical and horizontal split of tuples with bolts. With more realistic data would a tuple ID be needed, and could also be used to guarantee no duplicate tuples in skyline.
- With a more realistic approach would defining a set of realistic constraints on dimensions make filter bolts more useful.
- Flushing of the tumble window with the use of tick tuples. Achieving a more realistic handling of window in contrast to restarting topologies every X minute.
- Improving topologies with better strategies for filter bolts, both on the decision of filter points and possible sharing of them.
- Making use of the upcoming Storm 9.+, and utilize the pluggable worker communication. Making use of Netty [19] looks promising as it is a pure Java based implementation, that most likely will give a huge boost in worker to worker communication.
- Utilizing of Storms pluggable scheduler for achieving fine grained control over which executors that are assigned to which nodes.

References

[1] About storm: Free and open source. http://storm-project.net/about/free-and-open-source.html. Accessed: 07/03/2013.

[2] About storm: Guarantees data processing. http://storm-project.net/about/guarantees-data-processing.html. Accessed: 11/03/2013.

[3] About storm: Integrates. http://storm-project.net/about/integrates.html. Accessed: 05/05/2013.

[4] About storm: scalable. http://storm-project.net/about/scalable.html. Accessed: 09/05/2013.

[5] About storm: Simple-api. http://storm-project.net/about/simple-api.html. Accessed: 05/05/2013.

[6] About storm: Use with any language. http://storm-project.net/about/multi-language.html. Accessed: 07/03/2013.

[7] The apache cassandra project. http://cassandra.apache.org/. Accessed: 09/03/2014.

[8] Apache thrift - wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Apache_Thrift. Accessed: 20/02/2014.

[9] Complex event processing: Dsl for high frequency trading. http://www.infoq.com/presentations/DSL-for-High-Frequency-Trading. Accessed: 28/10/2013.

[10] Concepts · nathanmarz/storm wiki · github. https://github.com/nathanmarz/storm/wiki/Concepts. Accessed: 07/05/2013.

[11] Disruptor by lmax-exchange. http://lmax-exchange.github.io/disruptor/. Accessed: 04/03/2014.

[12] Distributed-rpc. https://github.com/nathanmarz/storm/wiki/Distributed-RPC. Accessed: 24/07/2013.

[13] Esotericsoftware/kryo · github. https://github.com/EsotericSoftware/kryo. Accessed: 17/02/2014.

[14] Fault tolerance. https://github.com/nathanmarz/storm/wiki/Fault-tolerance. Accessed: 07/03/2013.

[15] Grid network. http://en.wikipedia.org/wiki/Grid_network. Accessed: 20/03/2014.

[16] Hadoop. http://en.wikipedia.org/wiki/Hadoop. Accessed: 12/04/2013.

[17] Ibm what is big data? `http://www-01.ibm.com/software/data/bigdata/`. Accessed: 27/02/2013.

[18] Mapreduce. `http://wiki.apache.org/hadoop/MapReduce`. Accessed: 05/02/2013.

[19] Netty: Home. `http://netty.io/index.html`. Accessed: 17/02/2014.

[20] Powered by. `https://github.com/nathanmarz/storm/wiki/Powered-By`. Accessed: 27/10/2013.

[21] Rabbitmq - what can rabbitmq do for you? `https://www.rabbitmq.com/features.html`. Accessed: 17/02/2014.

[22] Rationale. `https://github.com/nathanmarz/storm/wiki/Rationale`. Accessed: 13/05/2013.

[23] Riak | basho technologies. `http://basho.com/riak/`. Accessed: 09/03/2014.

[24] S4 distributed stream computing platform. `http://incubator.apache.org/s4/`. Accessed: 08/02/2013.

[25] S4 project incubation status. `http://incubator.apache.org/projects/s4.html`. Accessed: 27/10/2013.

[26] Setting up a storm cluster. `https://github.com/nathanmarz/storm/wiki/Setting-up-a-Storm-cluster`. Accessed: 08/06/2013.

[27] Storm. `http://storm-project.net/index.html`. Accessed: 30/08/2013.

[28] Storm 0.8.0 released. `http://storm-project.net/2012/08/02/storm080-released.html`. Accessed: 04/12/2013.

[29] Storm 0.8.2 released. `http://storm-project.net/2013/01/11/storm082-released.html`. Accessed: 15/07/2013.

[30] [storm-68] need multiple output streams for a bolt in trident - asf jira. `https://issues.apache.org/jira/browse/STORM-68`. Accessed: 24/02/2014.

[31] Storm api - config. `http://nathanmarz.github.io/storm/doc/backtype/storm/Config.html`. Accessed: 26/02/2014.

[32] Storm api - irichbolt. `http://nathanmarz.github.io/storm/doc/backtype/storm/topology/IRichBolt.html`. Accessed: 18/02/2014.

[33] Storm api - irichspout. `http://nathanmarz.github.io/storm/doc/backtype/storm/topology/IRichSpout.html`. Accessed: 02/12/2013.

[34] Storm: Project ideas. `https://github.com/nathanmarz/storm/wiki/Project-ideas`. Accessed: 24/05/2013.

[35] Storm project incubation status. `http://incubator.apache.org/projects/storm.html`. Accessed: 07/03/2013.

[36] Storm: Structure of the codebase. `https://github.com/nathanmarz/storm/wiki/Structure-of-the-codebase`. Accessed: 11/10/2013.

[37] Storm wiki - documentation. `https://github.com/nathanmarz/storm/wiki/Documentation`. Accessed: 18/02/2014.

[38] Storm wiki - structure of the codebase. `https://github.com/nathanmarz/storm/wiki/Structure-of-the-codebase`. Accessed: 18/02/2014.

[39] storm/changelog.md at master · nathanmarz/storm. `https://github.com/nathanmarz/storm/blob/master/CHANGELOG.mdl`. Accessed: 09/10/2013.

[40] Topologybuilder. `http://nathanmarz.github.io/storm/doc-0.8.1/backtype/storm/topology/TopologyBuilder.html`. Accessed: 20/02/2014.

[41] Twitter storm: Open source real-time hadoop. `http://www.infoq.com/news/2011/09/twitter-storm-real-time-hadoop`. Accessed: 07/03/2013.

[42] twitter/kestrel · github. `https://github.com/twitter/kestrel`. Accessed: 16/02/2014.

[43] Welcome to apache™ hadoop®! `http://hadoop.apache.org/`. Accessed: 14/04/2013.

[44] Ømq. `http://zeromq.org/`. Accessed: 16/02/2014.

[45] Ømq java binding. `http://zeromq.org/bindings:java`. Accessed: 16/02/2014.

[46] S. Borzsony, D. Kossmann, and K. Stocker. The skyline operator. In *Data Engineering, 2001. Proceedings. 17th International Conference on Data Engineering*, pages 421–430, 2001.

[47] Lijiang Chen, Bin Cui, and Hua Lu. Constrained skyline query processing against distributed data sites. *Knowledge and Data Engineering, IEEE Transactions on*, 23(2):204–217, 2011.

[48] Christos Doulkeridis and Kjetil Nørvåg. A survey of large-scale analytical query processing in mapreduce. *The VLDB Journal*, pages 1–26, 2013.

[49] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 78–91, New York, NY, USA, 1997. ACM.

[50] Trisha Gee. Trisha's ramblings. `http://mechanitis.blogspot.no/2011/06/dissecting-disruptor-whats-so-special.html`. Accessed: 14/02/2014.

[51] Lukasz Golab and M. Tamer Özsu. Issues in data stream management. *SIGMOD Rec.*, 32(2):5–14, June 2003.

[52] Katja Hose and Akrivi Vlachou. A survey of skyline processing in highly distributed environments. *The VLDB Journal*, 21(3):359–384, June 2012.

[53] EsperTech Inc. Esper faq. `http://esper.codehaus.org/tutorials/faq_esper/faq.html`. Accessed: 12/03/2013.

[54] Jonathan Leibiusky, Gabriel Eisbruch, and Dario Simonassi. *Getting Started with Storm.* O'Reilly Media Inc., September 2012.

[55] Nathan Marz. A storm is coming: more details and plans for release. `http://engineering.twitter.com/2011/08/storm-is-coming-more-details-and-plans.html`, August 2011. Accessed: 08/02/2013.

[56] Michael G. Noll. Understanding the internal message buffers of storm - michael g. noll. `http://www.michael-noll.com/blog/2013/06/21/understanding-storm-internal-message-buffers/`. Accessed: 05/02/2014.

[57] Michael Steinbach Pang-Ning Tan and Vipin Kumar. *Introduction to Data Mining.* Pearson Education, Inc, 2006.

[58] J. Shore. Fail fast [software debugging]. *Software, IEEE*, 21(5):21–25, 2004.

[59] Shengli Sun, Zhenghua Huang, Hao Zhong, Dongbo Dai, Hongbin Liu, and Jinjiu Li. Efficient monitoring of skyline queries over distributed data streams. *Knowledge and Information Systems*, 25(3):575–606, 2010.

[60] Martin Thompson and Michael Barker. Lmax - how to do 100k tps at less than 1ms latency. `http://www.infoq.com/presentations/LMAX`. Movie time: 36:00. Accessed: 15/02/2014.

[61] Richard Tibbetts. Streambase high availability. `http://www.StreamBase.com/wp-content/uploads/downloads/StreamBase_White_Paper_High_Availability.pdf`.

[62] Richard L. Villars, Matthew Eastwood, and Carl W. Olofsen. Big data: What it is and why you should care. `http://sites.amd.com/us/Documents/IDC_AMD_Big_Data_Whitepaper.pdf`, June 2011.

[63] Jin xian Lin and Jing jing Wei. Constrained skyline computing over data streams. In *e-Business Engineering, 2008. ICEBE '08. IEEE International Conference on*, pages 155–161, Oct 2008.

9 Appendix: Details

Note:

When commands are listed in subsections, and nothing else is specified are those commands to be interpret as commands that is supposed to be ran in a Linux shell (a command-line interpreter).

Exception:

- # indicates comments and will not execute anything.
- **vim** is a text editor, the edit to be done is indicated with *:::VIM:::* Some text edit. *:::END:::*

9.1 Setup of local Storm cluster

”Storm has a ’local mode’ where a Storm cluster is simulated in-process. This is useful for development and testing.” [5] This section will cover how to set up a local Storm cluster for local topology testing and building Storm projects using Storm, Java, Clojure and Leiningen.

9.1.1 Make the workstation ready

To be able to use Storm locally do we need to download a copy of Storm and extract it. We would also need to install leiningen, it would be our build tool for creating jar archives. The installation can be done with the following commands:

Listing 7: ’Install Storm on workstation.’

```
cd ~
wget https://www.dropbox.com/s/fl4kr7w0oc8ihdw/storm-0.8.2.zip
unzip storm-0.8.2.zip
sudo apt-get install leiningen -y
```

9.1.2 Compiling and packaging of dependencies

Leiningen is a build and dependency management tool. It is used for automating Clojure projects. The aim of *Leiningen* is offering a build tool with less complexity than *Maven*. *Leiningen* is invoked with the following command: *”lein arguments”*

Most used commands arguments are:

- **jar** - Compile and package up all the project’s files into a jar file.
- **javac** - Compile Java source files.
- **uberjar** - Compile and package up the project files and all dependencies into a jar file.
- **help** - Displaying the tools CLI help.

9.1.3 Setting up a fresh storm project

First thing needed is a project workspace:

Listing 8: 'Setting up a fresh Storm project.'

```
$ mkdir -p ~/project/src/no/randompackage
```

Second, a definition of the project. Name it '*project.cli*' and save it in '*∼/project/*' folder.

Listing 9: 'project.clj'

```
;; ~/project/project.clj
(defproject storm-starter "0.0.1-SNAPSHOT"
  :java-source-path "src/"
  :javac-options {:debug "true" :fork "true"}
  :jvm-opts ["-Djava.library.path=/usr/local/lib:/opt/local/lib:/usr/lib"]
  :dependencies []
;;  :main [TopologyMain]
  :dev-dependencies [
                     [storm "0.8.2"]
                     ])
```

Implement a test topology in the folder '*∼/project/src*'. Section 5 can be used as a starting point.

To create an jar containing the dependencies and the topology. Run the following command while standing in the project root ('*∼/project/*):

Listing 10: 'Compile and create a jar.'

```
lein uberjar
```

To launch Storm topology defined in TopologyMain.class, run the following command:

Listing 11: 'Launch Storm topology on workstation.'

```
# Launch storm with given topology
storm-0.8.2/bin/storm jar topology.jar no\.randompackage.\TopologyMain
```

9.1.4 Error probing

When Storm crashes it is wise to check the log directories for clues. Storm logs for workers, nimbus and ui is located in '∼/storm-0.8.2/logs/*' When Storm is ran in remote cluster mode, those logs will be located on their representative nodes. Logs related to supervising of Storm components, are located at representative nodes in the folder '*∼/log/storm/**'

It is worth mentioning, when Storm can not execute a given topology, Storm will not do anything. More troubleshooting information can also be found at: `https://github.com/nathanmarz/storm/wiki/Troubleshooting`

9.1.5 Fault handling

Storm is designed to be fault safe. It keeps no track of state in Storm Supervisor, workers or nimbus process. The required state info is kept safe in Zookeeper cluster. When Storm components encounters errors and fails, they will die visible. It is the supervisors task to relaunch a failing component.

9.2 Setup of remote Storm cluster

For the workstation to connect and submit jars containing topologies and resources to a remote Storm cluster, it needs to know where the nimbus instance is located. To define for Storm CLI where nimbus is located run the following:

```
mkdir ~/.storm;
echo 'nimbus.host: "10.0.0.212"' >> ~/.storm/storm.yaml
```

9.2.1 Requirements for Storm cluster

To set up a recommended Storm cluster, the following is needed; An Apache Zookeeper cluster 3.3.3, ∅MQ 2.1.7, J∅MQ for ∅MQ 2.1.7, Storm 0.8.2, Java 6 and Python 2.6.6. The ZooKeeper cluster is used to manage the necessary state info for the different components of the cluster. J∅MQ is the Java binding for ∅MQ, which is used as the worker to worker messaging passing system [26]. I have successfully ran Storm with Zookeeper 3.3.5 and Oracle Java 7.

9.2.2 Setup Storm Clusters

It is possible to automatically deploy and configure Storm on Amazon Web Services EC2 with the storm-deploy project [26].

To set up a supervisor node on Ubuntu 12.04.3 the following can be used as a guide:

"Setup Storm Supervisor"

```
# This is not a bash/sh script but a listing of commands, with
# highlighting in certain editors, for installing Storm Supervisor cluster
# and it depencies on Ubuntu LTS 12.04.3. Locale used 'en_US.UTF-8'.
#
# Installed with 'server' and 'openssh-server' package
# Can be installed with:
# sudo tasksel install server
# sudo tasksel install openssh-server

# Note: 10.0.0.209 is hosting the Zookeeper-server, and Storm nimbus.
# ØMQ, JØMQ and the build dependencies tools should also be installed
# on the Nimbus and UI node.

# Upgrade last kernel minior release.
sudo apt-get update && sudo apt-get dist-upgrade -y

# If a newer kernel is needed due to newer hardware
# sudo apt-get install linux-generic-lts-saucy -y

# Install build-depencies tools services.
sudo apt-get install make g++ openjdk-6-jdk automake git -y
sudo apt-get install pkg-config libtool supervisor tree -y
sudo apt-get install uuid-dev libpgm-5.1-0 autoconf unzip -y

# Set JAVA_HOME for buildtools to find java.
export JAVA_HOME=/usr/lib/jvm/java-6-openjdk-amd64/

# Download build and install ZeroMQ.
cd ~
wget http://download.zeromq.org/zeromq-2.1.7.tar.gz
tar -xzf zeromq-2.1.7.tar.gz
cd ~/zeromq-2.1.7
./autogen.sh
./configure
make
sudo make install

# Download build and install Javabindings for ZeroMQ.
cd ~
```

```
git clone https://github.com/nathanmarz/jzmq.git
cd ~/jzmq/src
touch classdist_noinst.stamp
CLASSPATH=.:./.:$CLASSPATH javac -d . org/zeromq/ZMQ.java
    org/zeromq/ZMQException.java org/zeromq/ZMQQueue.java
    org/zeromq/ZMQForwarder.java org/zeromq/ZMQStreamer.java
cd ~/jzmq/
./autogen.sh
./configure
make
sudo make install

# Download, install and configure Storm.
cd ~
wget https://www.dropbox.com/s/fl4kr7w0oc8ihdw/storm-0.8.2.zip
unzip storm-0.8.2.zip
vim storm-0.8.2/conf/storm.yaml
:::VIM:::
storm.zookeeper.servers:
     - "10.0.0.209"
nimbus.host: "10.0.0.209"
# The number of workers running on the machine, for each port is a
    available worker.
supervisor.slots.ports:
    - 6700
#    - 6701
#    - 6702
#    - 6703
storm.local.dir: "/mnt/storm"
:::END:::

# Create the Storm datafolder and give it correct permissions.
sudo mkdir -p /mnt/storm
sudo chown stormuser:stormuser /mnt/storm

# Let the stormuser controll the supervisor
sudo vim /etc/supervisor/supervisord.conf
:::VIM:::
chown=root:supervisor
chmod=0770
:::END:::
sudo groupadd supervisor
sudo service supervisor restart
sudo usermod -a -G supervisor stormuser

# Put storm supervisor under supervisoring and let it access log
sudo mkdir -p /home/stormuser/log/storm/
sudo chmod -R 775 /home/stormuser/log/
sudo chown -R root:stormuser /home/stormuser/log
sudo vim /etc/supervisor/conf.d/storm_supervisor.conf
```

```
:::vim:::
[program:storm_supervisor_script]
user=stormuser
command=/home/stormuser/storm-0.8.2/bin/storm supervisor
autostart=true
autorestart=true
stderr_logfile=/home/stormuser/log/storm/supervisor.err.log
stdout_logfile=/home/stormuser/log/storm/supervisor.out.log
:::END:::

# Make sure the Storm supervisor can reach each other.
sudo vim /etc/hosts
:::VIM:::
# NB! If the dns ain't having localhostnames mapped to IP, all
# IPs and corresponding hostname will be needed to manually
# add to the hosts file.
:::END:::

# Make the supervisor is started, load new config and starts
# supervisoring the Storm supervisor.
sudo service supervisor restart
sudo supervisorctl reload


# ifstat -S have been used to show current network traffic on
# spesific hosts over ssh.
sudo apt-get install ifstat
```

Setup Zookeeper cluster

A Storm cluster use one Zookeeper cluster for its internal coordination of nodes. The load on the Zookerper cluster is low, as Zookeeper is not used for internal message passing. In most cases a singel node Zookeeper cluster, will be sufficient. However for supporting failover or large Storm custers a cluster should be used [26]. Zookeeper is fail-fast, like Storm components, and will exit on errors. That implies that Zookeeper should also be run under a supervisor service. When running Zookeeper in a production environment, it is critical to set up some handling of Zookeepers massive data and transaction logs. That can be done with a *cron* task. *Cron* is a daemon to execute scheduled commands. The linux command 'crontab -e' can be used to set up a cron task.

Installing and setup of Zookeeper is outlined bellow:

"Setup Zookeeper"

```
# This is not a bash/sh script but a listing of commands, with
# highlighting in certain editors, for installing Zookeeper single node
    cluster
# and it depencies on Ubuntu LTS 12.04.3. Locale used 'en_US.UTF-8'.

sudo apt-get install zookeeper supervisor -y

# Let the zoouser controll the supervisor
sudo vim /etc/supervisor/supervisord.conf
:::VIM:::
chown=root:supervisor
chmod=0770
:::END:::
sudo groupadd supervisor
sudo usermod -a -G supervisor zoouser
sudo usermod -a -G zookeeper zoouser

# Put Zookeeper under supervisoring and let it access log
sudo mkdir -p /home/zoouser/log/zookeeper/
sudo chmod -R 775 /home/zoouser/log/
sudo chown -R root:zoouser /home/zoouser/log
sudo vim /etc/supervisor/conf.d/zookeeper.conf
:::vim:::
[program:zookeeper_script]
user=zookeeper
command=/usr/share/zookeeper/bin/zkServer.sh start-foreground
autostart=true
autorestart=true
stderr_logfile=/home/zoouser/log/zookeeper/zookeeper.err.log
stdout_logfile=/home/zoouser/log/zookeeper/zookeeper.out.log
:::END:::

# Changes to the standard Zookeeper configuration can be done here
sudo vim /etc/zookeeper/conf/zoo.cfg
:::vim:::
# specify all zookeeper servers
server.1=zookeeperHostIP:2888:3888
# On a single node cluster should the leader serve clients.
leaderServes=yes
:::END:::

# Set the zookeeper nodeID in the zookeeper cluster.
sudo vim /etc/zookeeper/conf/myid

# Make the supervisor is started, load new config and starts
# supervisoring the Storm supervisor.
sudo service supervisor restart
sudo supervisorctl reload
```

Setting up Storm Nimbus and Storm UI

Follow the outlining bellow to install Storm Nimbus and Storm UI onto one node.

"Storm Nimbus and Storm UI"

```
# This is not a bash/sh script but a listing of commands, with
# highlighting in certain editors. Its purpose is installing Nimbus node
    with Storm UI
# and its dependencies on Ubuntu LTS 12.04.3. Locale used 'en_US.UTF-8'.

# ØMQ, JØMQ and the build dependencies tools should also be installed on
# the Nimbus and UI node. How to do it is covered under, Setup Storm
    Supervisor.


# Download, install and configure Storm.
cd ~
wget https://www.dropbox.com/s/fl4kr7w0oc8ihdw/storm-0.8.2.zip
unzip storm-0.8.2.zip
vim storm-0.8.2/conf/storm.yaml
:::VIM:::
storm.zookeeper.servers:
     - "10.0.0.209"
nimbus.host: "10.0.0.209"
#supervisor.slots.ports:
#    - 6700
#    - 6701
#    - 6702
#    - 6703
storm.local.dir: "/mnt/storm"
:::END:::

# Create the storm datafolder and give it correct permissions.
sudo mkdir -p /mnt/storm
sudo chown stormuser:stormuser /mnt/storm

# Let the stormuser controll the supervisor
sudo vim /etc/supervisor/supervisord.conf
:::VIM:::
chown=root:supervisor
chmod=0770
:::END:::
sudo groupadd supervisor
sudo service supervisor restart
sudo usermod -a -G supervisor stormuser

# Put storm supervisor under supervisoring and let it access log
sudo mkdir -p /home/stormuser/log/storm/
sudo chmod -R 775 /home/stormuser/log/
sudo chown -R root:stormuser /home/stormuser/log
sudo vim /etc/supervisor/conf.d/storm_nimbus.conf
:::vim:::
[program:storm_nimbus_script]
user=stormuser
command=/home/stormuser/storm-0.8.2/bin/storm nimbus
autostart=true
autorestart=true
stderr_logfile=/home/stormuser/log/storm/nimbus.err.log
stdout_logfile=/home/stormuser/log/storm/nimbus.out.log
:::END:::

# Put Storm UI under supervisoring and let it access log
sudo vim /etc/supervisor/conf.d/storm_ui.conf
:::vim:::
[program:storm_ui_script]
user=stormuser
command=/home/stormuser/storm-0.8.2/bin/storm ui
autostart=true
autorestart=true
stderr_logfile=/home/stormuser/log/storm/ui.err.log
stdout_logfile=/home/stormuser/log/storm/ui.out.log
:::END:::

# Make the supervisor is started, load new config and starts
# supervisoring the Storm supervisor.
sudo service supervisor restart
sudo supervisorctl reload
```

9.3 Storm CLI

Coverage of available Storms Command Line Interface commands.

”Storm CLI”

```
# Execute a topology
storm jar storm-packed-topology.jar topology\.package\.path\topologyclass
    argument1 argumentN

# Kill a topology, and allows it to wait x seconds for current processing
    messages to finish.
storm kill topology-name [-w wait-time-secs]

# Activate a topology's spouts.
storm activate topology-name
```

```

# Deactivate a topology's spouts.
storm deactivate topology-name

# Rebalance a topology automatic onto new nodes, keep the number of workers
    and current parallelism.
storm rebalance topology-name [-w wait-time-secs]

# Rebalance a topology manually, changing the number of workers and current
    parallelism of components. The e parameter is changing the number of
    Executors for the given bolt\spout.
storm rebalance topology-name [-w wait-time-secs] -n new-number-of-workers
    -e spout-name=new-number -e bolt-name=new-number

# Open a Clojure REPL, with the storm jars and configuration on classpath.
storm repl

# Print Storms clients classpath.
storm classpath

# Print the value of a entry in the local Storm config.
storm localconfvalue conf-name

# Print the value of a entry in the Storm cluster's config. Must be run on
    a cluster machine.
storm remoteconfvalue conf-name

# Starts the Storm Nimbus, should be run under supervisor service.
storm nimbus

# Starts the Storm Supervisor, should be run under supervisor service.
storm supervisor

# Starts the Storm UI webserver, should be run under supervisor service.
storm ui

# Starts the Storm DRPC, should be run under supervisor service.
storm drpc
```

9.4 Java Virtual Machines

To use Oracle 7 JVM instead of OpenJDK the following is needed; Download Oracle Java Runtime Environment (JRE) or Oracle Java Development Kit (JDK) from `http://www.oracle.com/technetwork/java/javase/downloads/index.html` . Next follow the outlining bellow, it is necessary to adopt commands to your downloaded file.

”Changing java.”

```
# Extract the and change owner.
sudo tar xzvf jdk-7u45-linux-x64.tar.gz -C /usr/lib/jvm/
sudo chown -R root:root /usr/lib/jvm/jdk1.7.0_45

# Tell Ubuntu it exists.
sudo update-alternatives --install "/usr/bin/java" "java"
    "/usr/lib/jvm/jdk1.7.0_45/bin/java" 1

# Pick the desired Java to use.
sudo update-alternatives --config java
```

The Java JRE is not containing the developer tools and *javac*, that is used for compiling. Installing and changing the default *javac* command can be done the same way.

”Changing javac.”

```
# Make sure the JDK, is extracted and have correct ownership.

# Tell Ubuntu it exists.
sudo update-alternatives --install "/usr/bin/javac" "javac"
    "/usr/lib/jvm/jdk1.7.0_45/bin/javac" 1

# Pick the desired Javac to use.
sudo update-alternatives --config javac
```

9.5 Changing Storm Parallelism on the fly

An already running topology can have its bolt parallelism changed without restarting cluster or resubmitting Storm topology to cluster. Storm UI can issue an auto rebalance command. Storm CLI tool can submit an extensive rebalance. That rebalance allow changing number of workers assigned to a topology, and number of executors assigned to a bolt or spout. Number of tasks is permanent and can not be changed on the fly.

”Storm rebalance.”

```
# Rebalance a topology manually, changing the number of workers and current
    parallelism of components.
storm rebalance topology-name -w wait-time-secs -n new-number-of-workers -e
    spout-name=new-number -e bolt-name=new-number -e
    other-bolt-name=new-number

# An example rebalance of the "Example topology code" in Section  5.3.3 on
    page  30.
storm rebalance exampleTopology -w 10 -n 4 -e SpoutTwo=64 -e BoltOne=1
```

When rebalance command is issued, topology will deactivate spouts, wait 10 seconds and then carry out the changes. After rebalance parallelism will be; Four worker nodes utilized, SpoutOne will not be touched, SpoutTwo will get 64 executors and BoltOne will still have one executor.

9.6 Troubleshooting

As of Storms current state, it is nearly impossible to stay out of running into errors. Errors that to some extent can be obfuscated compared to user friendly ones. This part will cover some of those errors encountered and feasible solutions.

Failing to submit topology to a nimbus cluster.

”Thrift error”

```
java.lang.RuntimeException: org.apache.thrift7.transport.TTransportException: java.
    net.ConnectException: Connection refused
        at backtype.storm.utils.NimbusClient.<init>(NimbusClient.java:36)
        at backtype.storm.utils.NimbusClient.getConfiguredClient(NimbusClient.java
            :17)
        at backtype.storm.StormSubmitter.submitTopology(StormSubmitter.java:69)
        at backtype.storm.StormSubmitter.submitTopology(StormSubmitter.java:40)
        at no.stud.BlackHoleTopology.main(BlackHoleTopology.java:90)
Caused by: org.apache.thrift7.transport.TTransportException: java.net.
    ConnectException: Connection refused
        at org.apache.thrift7.transport.TSocket.open(TSocket.java:183)
        at org.apache.thrift7.transport.TFramedTransport.open(TFramedTransport.
            java:81)
        at backtype.storm.utils.NimbusClient.<init>(NimbusClient.java:34)
        ... 4 more
Caused by: java.net.ConnectException: Connection refused
        at java.net.PlainSocketImpl.socketConnect(Native Method)
```

Possible solution:

This error is caused when Storm client cant reach nimbus service. First check network and that nimbus service is running.

Make sure it is only one entry for *nimbus.host* in '*~/.storm/storm.yaml*' file. If no file exist, it can be crated with the following commands.

```
mkdir ~/.storm;
echo 'nimbus.host: "10.0.0.212"' > ~/.storm/storm.yaml
```

Make sure to substitute the IP-address with the correct IP-address assigned to Nimbus host. Further notice; Do not interpret '*~/.storm/storm.yaml*' to be the same file or interchangeable with the '*storm/conf/storm.yaml*' file. The later is used by the Nimbus and Supervisor daemons. Storm is known not to be compatible with IPv6, make sure only IPv4 is available.

IPv6

Storm is not compatible with IPv6.

Possible solution:

Possible solutions is to disable IPv6 for the interface Storm is assigned to (e.g. interface eth0). It is also possible to totally disable IPv6. After changing the referenced config file, do a reboot.

IPv4 can be enforced for storm supervisor children from the configuration, adding *-Djava.net.preferIPv4Stack=true* to the supervisor child options and restarting the supervisor should work. *-Djava.net.preferIPv4Stack=true* can also be set directly from Java with the System.setProperty("java.net.preferIPv4Stack", "true");

I preferred turning IPv6 off directly on the interface.

"Disable IPv6 on eth0:"

```
#
# /etc/sysctl.conf
# Add the following line to the .conf for
# disabling IPv6 for eth0.
#
net.ipv6.conf.eth0.disable_ipv6 = 1
```

"Disable IPv6 totally:"

```
#
# /etc/sysctl.conf
# Add the following lines to the file for
# disabling IPv6 for all interfaces.
#
net.ipv6.conf.all.disable_ipv6 = 1
net.ipv6.conf.default.disable_ipv6 = 1
net.ipv6.conf.lo.disable_ipv6 = 1
```

Failing to start supervisor:
Supervisor restart after a few secounds:

"Possible error message from supervisor.log"

```
2014−02−09 22:03:44 supervisor [INFO] Starting supervisor with id 3a042ed7−54bf−4
    ddc−8e48−15668a6a3230 at host i5
2014−02−09 22:03:45 event [ERROR] Error when processing event
java.lang.RuntimeException: java.io.EOFException
        at backtype.storm.utils.Utils.deserialize(Utils.java:68)
        at backtype.storm.utils.LocalState.snapshot(LocalState.java:24)
        at backtype.storm.utils.LocalState.get(LocalState.java:28)
        at backtype.storm.daemon.supervisor$sync_processes.invoke(supervisor.clj:192)
        at clojure.lang.AFn.applyToHelper(AFn.java:161)
        at clojure.lang.AFn.applyTo(AFn.java:151)
        at clojure.core$apply.invoke(core.clj:603)
        at clojure.core$partial$fn__4070.doInvoke(core.clj:2343)
        at clojure.lang.RestFn.invoke(RestFn.java:397)
        at backtype.storm.event$event_manager$fn__2507.invoke(event.clj:24)
        at clojure.lang.AFn.run(AFn.java:24)
        at java.lang.Thread.run(Thread.java:744)
Caused by: java.io.EOFException
        at java.io.ObjectInputStream$PeekInputStream.readFully(ObjectInputStream.
            java:2325)
        at java.io.ObjectInputStream$BlockDataInputStream.readShort(
            ObjectInputStream.java:2794)
        at java.io.ObjectInputStream.readStreamHeader(ObjectInputStream.java:801)
        at java.io.ObjectInputStream.<init>(ObjectInputStream.java:299)
        at backtype.storm.utils.Utils.deserialize(Utils.java:63)
        ... 11 more
```

Possible solution:

This error is caused when Storm Supervisor daemon is having a corrupted local state. It is known to run into such a state when topology or supervisor get an improper shut-down.

The solution is to clear up the supervisors local state. It can be done with the following command.

```
rm -rf /mnt/storm/supervisor/
```

9.7 Attached code

The complete code can be found in the attached .jar file.

”SimpleSkylineTopology”

```java
package no.stud;

import backtype.storm.Config;
import backtype.storm.LocalCluster;
import backtype.storm.StormSubmitter;
import backtype.storm.topology.TopologyBuilder;

import no.stud.bolts.SimpleSkylineBolt;
import no.stud.spouts.BinFileReadSpout;
import no.stud.spouts.GenerateDatasetSpout;
import no.stud.util.CustomConfig;
import no.stud.util.DistributionType;


public class SimpleSkylineTopology {
	public static void main(String[] args) throws InterruptedException {

	// Topology definition
		TopologyBuilder builder = new TopologyBuilder();
		String topologyName = "SimpleSkylineTopology";

		// GenerateData on the fly.
		// builder.setSpout("Generator", new GenerateDatasetSpout(),
		    3);

		// ReadDataset from file. /* N - how many instances */
		builder.setSpout("Generator",new BinFileReadSpout(), 1);

		builder.setBolt("Simple-SL", new SimpleSkylineBolt() , 1).
		    localOrShuffleGrouping("Generator");


	// Configuration, create a new configuration object that can be
	    sumbitted with the topology.
		Config conf = new Config();

		// Put arguments from ./storm jar file.jar Topology args1
		    argsn in configfile.
		// conf.put(submitArgs, args[0]);

		/** Data generator configuration. **/
		// Max value for a dimension in the point.
		// conf.put(CustomConfig.DimensionMaxValue, 1000000);
```

```

            // How many dimensions a point has.
            conf.put(CustomConfig.PointDimensions, 5);

            // What type of generator shall be used. /* Uniform,
                AntiCorrelated, Correlated */
            conf.put(CustomConfig.GeneratorType, DistributionType.
                ANTICORRELATED.getId());

            // Set the toggle sleep value, to enable to trottle the
                spouts.
            conf.put(CustomConfig.ToggleSleepValue, 2);

            // Set the sleep time in ms, default 1ms when enabled.
            conf.put(CustomConfig.ToggleSleepTime, 20);

            /** Topology configuration **/
            // Disabling reliability. (Setting how many executers to
                spawn for acking.)
            conf.put(Config.TOPOLOGY_ACKER_EXECUTORS, 0);

            // Bolts and Spouts will log eg. every tuppel they emit.
            conf.setDebug(false);

            // Maximum pending tuples, has no effect when reliability is
                disabled.
            conf.setMaxSpoutPending(5000);

            // Set the max number of storm workers. Remember acker is
                using one task.
            conf.setNumWorkers(1);

            // Fallback on java is extreme expencive and is there only
                for prototyping.
            conf.put(Config.TOPOLOGY_FALL_BACK_ON_JAVA_SERIALIZATION,
                false);

            // Make storm ignore kryo registrations that ain't availble
                on the classpath.
            conf.put(Config.TOPOLOGY_SKIP_MISSING_KRYO_REGISTRATIONS,
                true);

            // Register kyro serialization for the double[] datatype.
            conf.registerSerialization(double[].class);


            /** JVM options **/
            // Set a very aggresive GBC, as we mostly relay on generating
                 data and droping it, also tuning memory settings eg.
```

```
        conf.put(Config.TOPOLOGY_WORKER_CHILDOPTS, "-d64 -XX:GCTimeRatio=1 -XX:+AggressiveOpts -XX:+UseLargePages -XX:MaxNewSize=4g ");
        // internal note.
        // conf.put(Config.TOPOLOGY_WORKER_CHILDOPTS, "-Xmx768m -XX:MaxPermSize=1024m");

        // Deny config from automaticly getting some options two times.
        conf.put(Config.WORKER_CHILDOPTS, " ");

        /** Nimbus fails heartbeat **/
        // conf.put(conf.TOPOLOGY_EXECUTOR_RECEIVE_BUFFER_SIZE,128);
        // conf.put(conf.TOPOLOGY_EXECUTOR_SEND_BUFFER_SIZE,128);
        // conf.put(conf.NIMBUS_TASK_TIMEOUT_SECS,5);


        /** ZMQ (ZeroMQ), flow control parameters. ZMQ is fire and forget **/
        // Buffersize in items before, it takes action eg. blocking. Infinite
        // or a abitary high number will make it eat all your memory.  If all bolts can't keep up.
        // conf.put(Config.ZMQ_HWM, 0); // Controlling the behaivor in spots\bolts instead.
        // Number of ZMQ threads that should be used by the ZMQ context in each worker process.
        // conf.put(Config.ZMQ_THREADS, 1);
        // Setting the timeout for ZMQ
        // conf.put(Config.ZMQ_LINGER_MILLIS, 5000);


        // Decide on to run local or remote dipending on args.
        if (args != null && args.length == 0) {
            // Topology run cluster
            try{
                // Submit the topology.
                StormSubmitter.submitTopology(topologyName, conf, builder.createTopology());

            // Will be catched of storm and printed.
            } catch (Exception ex) {
                ex.printStackTrace();
                /*     Throws:
                       AlreadyAliveException - if a topology with this name is already running
                       InvalidTopologyException - if an invalid topology was submitted
                 */
            }
            } else if(args.length > 0) {

            // Local topology testing

                    // Local cluster config changes
                    conf.setMaxSpoutPending(1);                  // Maximum
                        pending tuples, has no effect when reliability is
                        disabled.
                    conf.setDebug(true);                         // Bolts
                        and Spouts will log eg. every tuppel they emit.

                    // Topology run local
                    LocalCluster cluster = new LocalCluster(); // Creat a
                        local cluster
                    cluster.submitTopology("LocalToplogie", conf, builder.
                        createTopology()); // Submit the topology to the
                        local cluster.

                    // Delay the shutdown of the local-cluster.
                    Thread.sleep(300000);                        // Delay in
                         ms. ~5 min.
                    cluster.shutdown();                          // Shut
                        down the local cluster.
            }
        } // end function
} // end class
```