Seminar Databasesystems

By
Samuel Kurath

University of Applied Science, HSR

Autumn 2016

Samuel Kurath

Advisor:
Prof. Keller Stefan

Seminar Databasesystems

Abstract

The content of this paper is splitted into three parts, it begins with an overview of data streams and their difficulties, followed by a part about Apache Storm a distributed stream processing framework. And finally a concrete implementation based on a given problem, solved with Apache Storm. The goal of the implementation is to analyze the minutely updated Augmented Diffs of OpenStreetMap and do a benchmark to test the performance of Apache Storm.

Contents

0 Introduction

Durring the Master of Science in Engineering the students have to participate two seminars. The goal of these is to elaborate a theme on their own, discuss the result in group and write a paper about the topic.

The Databasesystems Seminar does a focus on streams and how to do processing on them. Stream processing is a strong growing subject in reference to the huge amount of data we are exposed and produce these days.

A big force in generating data is the rapidly increasing amount of Internet of Things sensors, the willingness of the people to publish a lot of personal information on social media platforms and also the expanding interest in data collection of companies. Furthermore that leads to new business models with free products or services, but unfortunately there ain't no such thing as a free lunch, the private data you are sharing with these companies is the settlement.

With the huge amount of data new problems in collecting, processing and storing appear. Thus new solutions and technical tools to solve them appear too.

The man who is swimming against the stream
knows the strength of it.

Woodrow Wilson

1 Stream Processing

Steams are older than computers so it is not a big surprise that streams and the processing of them isn't a absolutly new topic in the computer science world. Historically there are techniques like **logging** or in the domain driven development area there is **event sourcing**, which are very similar to stream processing. But they have not been always as important as they are today with the unbelivable amount of data involved. The old way to handle the streams was an event driven one and to do analytics after storing the data.

Initial example

At this point I'd like to give you a small example of how streams and stream processing could be more and more relevant. And I would like to do this by the example of a factory producing delicious frosted yoghurt.

Frozen yogurt factory

Let me explain the *old* event driven way with a small example.

Imagine a factory which produces frozen yogurt in different flavors. They weigh and register every cup of yogurt at the end of the assembly line.

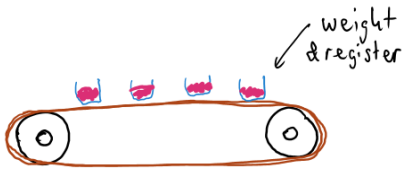

Figure 1.1: frozen yogurt assembly line

Common architecture

The factory has sensors which weigh the cups and the weights are sent to a server. The server handles the request and stores the frozen yogurt with his weight in the database. For analytic tasks there is a web application. If you are now interested in the total amount of produced cups. You can simply open the browser go to the analytic page. Which starts a request to the server and the server will call the database with a quey like "select count(*) from frozen_yogurt". After the quey is executed you will get the result from the server and have the aggregated number on your screen.

(This is more or less a default example of a Three-tier architecture)

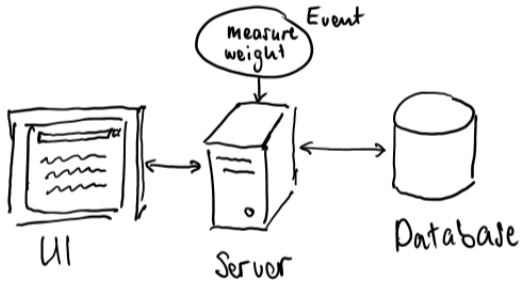

Figure 1.2: Three-tier architecture

Now the owner of the factory does a very good business and they are able to expand the production. And with the expansion they add a lot more of sensors to the assembly line, like a temperature sensor, optical recognition to check if the cups are always full and a lot more.

The requirements of the system are updated too, the owner wants to have statistical information about the production all the time and immediately notifications if for example the temperature is to high.

These new requirements leads to new challenges in the architecutre of the system and are hard to implement with the current state and the enormus data produced by all the sensors.

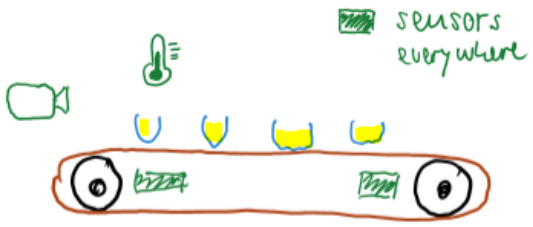

Figure 1.3: Sensors everywhere

Streaming architecture

A good way to handle this new requirements is to continuously aggregate and filter the stream of data before gets stored in the database. For this purpose there has grown up a lot of new techniques and frameworks during the last few years.

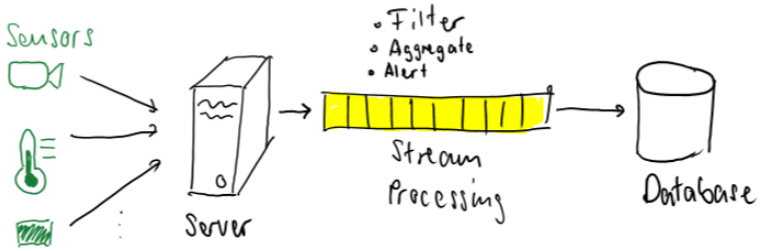

Figure 1.4: streaming architecture

Instead of the old three-tier architecture they added a message / logging queue like Apache Kafka to handle all concurrent access to data. Which enables a seamless access for the stream processing frameworks and the storing of the data for later analytics.

Conclusion

The new streaming architecture of the frozen yogurt factory has got a lot of advantages, but some big difficulties to handle too. Now it is possible to get very fast access to production statistical information, deal with the huge amount of data and throw alerts in real time. It's also helpfull in the developers perspective, because you can add several consumers to the stream. Normaly these streams are immutable, which solves the concurrent access of different consumers. Thus the requirements in such real time stream processing systems are ambitious and we will face them in the next section.

Eight Rules For Stream Procesing

In the paper "The 8 Requirements of Real-Time Stream Processing" Michael Stonebraker, Ugur Cetintemel and Stan Zdonik brought their thoughts about real-time stream processing together and wrote it down. It faces the fundamental ideas of stream processing and sums them up in eight rules which every stream processing engine should require. That allows us to measure and compare different technologies. Thus leads me to take a closer look at these eight rules and add my personal reflection to them.

Rule 1: Keep the Data Moving

The first requirement for a real-time stream processing system is to process messages in-stream, without any requirement to store them to perform any operation or sequence of operations. Ideally the system should also use an active (i.e., non-polling) processing model.

This is a very important point for me too. The stream processing should not slow down the "in-stream". How can this target be achieved? The paper raises the point of beeing aware of expensive storage operation. But I could imagine that this question depends most on the architecture of the processing engine. A solution from my point of view could be to handle the streams as immutable and decouple the processing from the data stream sink.

Rule 2: Query using SQL on Streams (StreamSQL)

The second requirement is to support a high-level StreamSQL language with built-in extensible stream- oriented primitives and operators.

Use something like "StreamSQL" is not a must have requirement in my eyes. I would even go further and say that a higher-level query language could not always fit the fast changing streams. So a flexible and easy to expand query language would be a better choice for me. Or if the maintainer are developers to realise the queries in code would be a good way too. Even in spite of the heavy maintenance and intelligibility.

Rule 3: Handle Stream Imperfections (Delayed, Missing and Out-of-Order Data)

The third requirement is to have built-in mechanisms to provide resiliency against stream imperfections, including missing and out-of-order data, which are commonly present in real-world data streams.

If the data of the stream does not appear like it used to be the stream processing engine should not crash and handle the problem. For me an other very fundamental requirement in the point of view that such streams often come from distributed systems. Which leads to a lot of hard to control error sources.

Rule 4: Generate Predictable Outcomes

The fourth requirement is that a stream processing engine must guarant predictable and repeatable outcomes.

The same input should always result in the same output. Easy said and easy to understand rule but in processing huge amount of data it could be in some cases very difficult. Imagine that there is more data then you are able to handle with all your computing power, but you still want to be able to process data and not every entry in the stream is absolutely important. Thus it could be a good strategy to more or less randomly throw some messages away. Nevertheless the decisions you do and the processing on it should be repeatable.

Rule 5: Integrate Stored and Streaming Data

The fifth requirement is to have the capability to efficiently store, access, and modify state information, and combine it with live streaming data. For seamless integration, the system should use a uniform language when dealing with either type of data.

I agree with this rule, past data and also current data should possible haven an influence on the results of the processing. So it is a good idea to use a uniform language. But I also think that the streaming engine needs to have the ability and flexibility to deal with different and unexpected data.

Rule 6: Guarantee Data Safety and Availability

The sixth requirement is to ensure that the applications are up and available, and the integrity of the data maintained at all times, despite failures.

Of course streaming engines need to be high-availability. But this is a very hard deal in the perspective of high performance, because high-availability leads to some inevitable overheads. Thus I would say that it should be possible to handle guarante integrity of mission-critical information and also support less critical information to be processed in a super performant but less integrity way.

Rule 7: Partition and Scale Applications Automatically

The seventh requirement is to have the capability to distribute processing across multiple processors and machines to achieve incremental scalability. Ideally, the distribution should be automatic and transparent.

In the case of huge amount of data the ability to scale vertically is an absolute must have feature of every stream processing engine. For me this is one of the most important rules.

Rule 8: Process and Respond Instantaneously

The eighth requirement is that a stream processing system must have a highly-optimized, minimal-overhead execution engine to deliver real-time response for high-volume applications.

Like at the rule 7 sad, this is a hard tradeoff between availability and integrity. Thus a stream processing engine has to deal with this two requirements as good as possible and find a best fitting solution. Because the possible throughput can also have a decisive impact on the choice of a technology.

There are some things you learn best in calm,
and some in storm.

Willa Cather

2 Apache Storm

Apache Storm is a reliable, distributed and fault-tolerant system for stream processing. The beginnings of the project were at Backtype (later bought by Twitter) and created by Nathan Marz. He open sourced Storm on September the 19th in 2011. The project rapidly got a big development coummunity and on September the 18, 2013 Nathan moved Storme to Apache Incubator.

Storm works with different types of components which are responsible for clear defined task. The components are bundled and managed in a so called **Topology**. The entrypoint and the stream input is handled by a **Spout**, the spout passes data to the **Bolts**. Bolts are responsible for the main data processing and persists the data. They can be chained or parallelised in a way that fits best for your current problem.

Figure 2.1: Storm

Topology

In generell Storm passes tuples between the different components. To organize the tuples there are topologies.

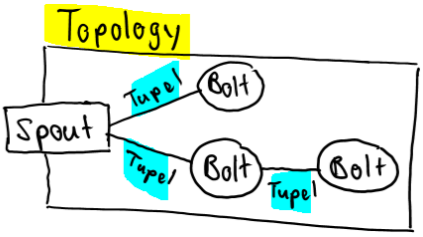

Figure 2.2: Topology example

Grouping

The Topology defines the grouping, this means how streams are consumed by the bolts and how they consume them. There are four kinds of grouping *Shuffle Grouping*, *Fields Grouping*, *All Grouping* and *Custom Grouping*.

Shuffle Grouping

Shuffle Grouping takes a single entry from the source and sends each tuple to a randomly choosen bolt, which is listening to this kind of tuples. It also garanties that each consumer gets more or less the same number of tuples.

Fields Grouping

With Fields Grouping it is possible to send tuples to spezified bolts based on the fields of the tuple. It takes care that the same combination of fields is always sent to the same bolt.

All Grouping

All Grouping sends every tuple to all the bolts. This is helpful for tasks like signals or to use different filters for alerting systems working on the same input.

Custom Grouping

It is possible to build your on grouping based on the stream. This is very helpful if you have to make fine granular desicions about which bolt has to handle which tuple.

Spout

The entry point of each topology are the so called spouts. Their main task is to procure the data from the input source. After the access of the source the spout emits a list of defined fields, which are consumed by the bolts. The spouts are also responsible for reliability, thus you have to take care to implement them in a fault-tolerant way. This means that a spout must have the ability to deal with unprocessable and unpredictable messages from the stream.

To handle the reliability at the spout you can add an ID to every message. If a message is correctly processed by all the target bolts the *ack* method of the spout is called. But if there were troubles or the timeout is reached the *fail* method will be triggered.

Example

As an example you can imagine a spout which consumes data from a message queue and emits the data to the interessted bolts. All of this data steam is defined in the Storm topology.

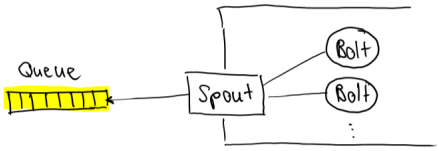

Figure 2.3: Spout example

Code

And in the code this could look like the following.

```
TopologyBuilder builder = new TopologyBuilder();
MessageQueueSpout spout = new MessageQueueSpout();
PrinterBolt bolt = new PrinterBolt();

builder.setSpout("spout", spout);
builder.setBolt("bolt", bolt).shuffleGrouping("spout");
```

Bolt

Bolts are the heartpieces, the engine of Apache Storm, they are responsible for the hard work of processing the data. A bolt takes tuples as input, does some operations on it and finally produces tuples as output. Normally a bolt implements the *IRichBolt* interface, which has the following methods.

Figure 2.4: Bolt engine

Methods

A Bolt provides the following basic methods.

1. void prepare(Map map, TopologyContext topologyContext)
2. void execute(Tuple tuple, BasicOutputCollector basicOutputCollector)
3. void declareOutputFields(OutputFieldsDeclarer outputFieldsDeclarer)
4. void cleanup()

prepare

This method is always called befor the bolts starts to process tuples (before the execute method)). A possible task could be to reset a counter.

execute

The execute methode processes a singel tuple of input, so it is the core function of every bold. So all the logic should be done here, for example filter a stream or counting the words of the input.

declareOutputFields

The prepare method is responsible for defining the output schema of the bolt. If bolts are concatenated the output is a tuple ,but if the current bolt is may be the last bolt in the chain, and does a task like, print to the standard output, you don't have to set an output type.

cleanup

Cleanup() is called when a bolt is going to shutdown. So you should do some tasks like a destructor does. For example disconnect an active database connection.

A good idea is about ten percent and implementation and hard work, and luck is 90 percent.

Guy Kawasaki

3 Implementation

The final chapter is about the implementation with Apache Storm to solve the defined queries and to do the benchmarks. The idea of the test scenario is to use "Augmented Diffs" from OpenStreetMap as a stream. The diffs extend the ordinary minutely diffs of OpenStreetMap with more information. The result contains all the nodes, ways and relations changed durring the time periode.

Test Setup

For the implementation there is a given test setup. The OpenStreetMap augmented diffs are minutelly collected by a script executed as a cron job. The script acts as an producer for Apache Kafka and the data is now accessible from these message queue. Furthermore the data is ready to be consumed by the stream processing engine and in the current case ready for Apache Storm.

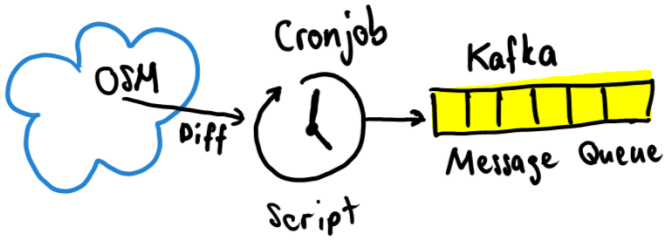

Figure 3.1: Test setup

Queries

The following list is about the queries which has to be done on the Augmented Diff stream.

A) Leader board of top 10 OSM active users

B) Leader board of top 10 OSM objects added

C) Node objects with suspicious keys and values

D) Way objects with only user tag "area=yes" without other user tags

Topology of the queries

With the queries we have to solve in mind, I have to design a fitting arcitecture for Apache Storm. So the first step is to get the messages from Kafka, then converte the message string to JSON, do the queries on the JSON stream and finally make the result available.

This leads me to the following Apache Storm topology:

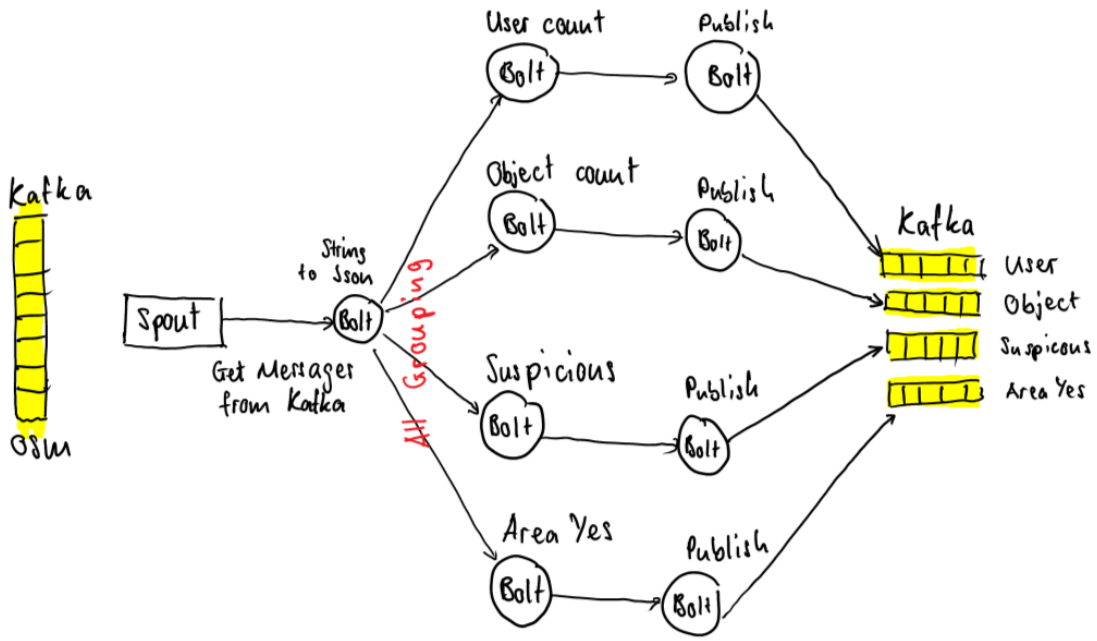

Figure 3.2: Topic for queries

Let's have a closer look to some implementation details of the image above. To read the messages from the Kafka osm topic, a topic is something lika a tabel in a relational database, I used the *KafkaSpout* class from org.apache.storm.kafka. This class handles the communication with Kafka and acts like a consumer, furthermore it makes the string messages availible for the bolts.

The consumed messages are in text format and needs to be casted to JSON for this purpose I implemented a *StringtoJsonBolt* class.

After this conversion I used an all grouping topology approach, which emits the JSON data to all connected bolts.

Every following bolt is responsible for the defined querie, thus leads to the classes *UserCountBolt*, *ObjectCountbolt*, *SuspicousBolt* and *AreaYesBolt*. When the bolts have done their queries on the JSON data there is always a connected *KafkaBolt* class from org.apache.storm.kafka, which finally publishs the results to the fitting Kafka topic.

And makes the data available for presentation tasks or further processing.

Benchmark

Benchmarking is a very difficult topic and strongly depends on various parameters like the underlying hardware. Thus we decide to make this as hardware independent as possible. The idea is to produce a lot of small message, like IoT does, and count all of them. Then repeat this process and relate the number of processed messages with the time spent.

For the message generation we had a small python script called "benchmark.py" which produces as much messages as you want and provide them with Apache Kafka. The messages have got the following structure:

packetId	TopicName	qos	retainFlag	payload	dupFlag
0	weather	5	False	83	True
1	computer	2	True	82	False

Table 3.1: MQTT Messages

To realize the benchmark with Apache Storm I tried two different topologies. The first with only one Bolt, which is the non parallelism way.

Figure 3.3: Benchmark 1 Bolt

For the second approach I used four Bolts for parallelism. The decision for four Bolts depends on the four cores of the notebook which I used for the benchmarks.

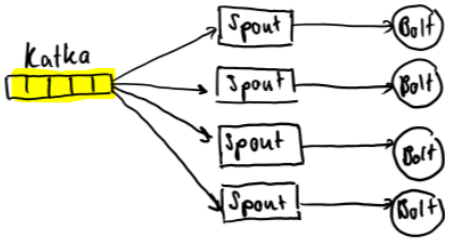

Figure 3.4: Benchmark 4 Bolts

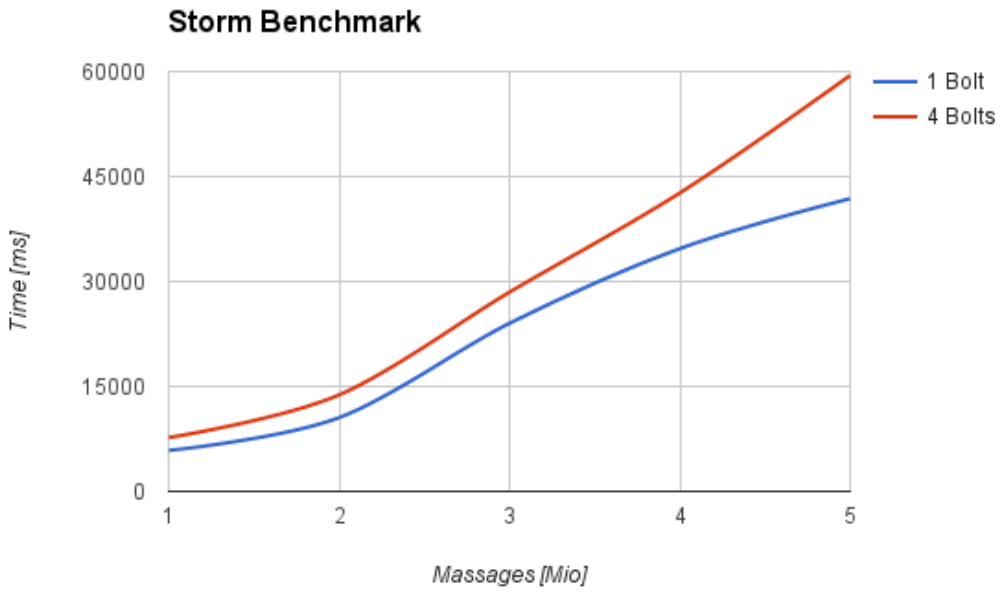

Figure 3.5: Benchmark Diagramm

The blue curve was the non parallel one with only one Bolt and the red was the parallel way with four Bolts.

Conclusion

To take a good conclusion out of this benchmark is a hard tasks. On the one hand side it's no problem for storm to process millions of messages. But on the other hand side I couldn't perform better with more messages. It was more or less a linear behavior. And a bit unsuspected was that the parallelism topic was slower than to work with only one Bolt. Maybe this behavior would change with bigger messages. I think the bottleneck in the construction was the communication with Kafka. And so the throughput was limited by the networking.

References

[1] Akidau, T., Balikov, A., Bekiroğlu, K., Chernyak, S., Haberman, J., Lax, R., McVeety, S., Mills, D., Nordstrom, P., & Whittle, S. (2013). Millwheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11), 1033–1044.

[2] Neumeyer, L., Robbins, B., Nair, A., & Kesari, A. (2010). S4: Distributed stream computing platform. In *2010 IEEE International Conference on Data Mining Workshops* (pp. 170–177).: IEEE.

[3] Stonebraker, M., Çetintemel, U., & Zdonik, S. (2005). The 8 requirements of real-time stream processing. *ACM SIGMOD Record*, 34(4), 42–47.

A

Installation

Repositories

For the implementation part I used different Github repositories to store the code. And help the user with the deployment and installation of the applications.

OSMstream

The OSMstream* repository provides a docker container which collects the augmented diffs from OpenStreetMap. For further use we convert the diffs from XML to JSON format and publish them with Apache Kafka.

Requirements:

- Scala 2.11
- Apache Kafka 0.10.0.1
- Python3
- Pip3
- or Docker >= 1.10

*https://github.com/geometalab/OSMstream

STORM

storm[†] is the main repository and hosts the Apache Storm implementation code with all the queries and a docker container for a simple usage. This application uses the data provided by OSMstream.

Requirements:

- OSMstream
- Apache Storm 1.0.2
- JVM >= 1.8
- Maven >= 3.3.3
- or Docker >= 1.10

StormBoard

The storm[‡] is a simple live streaming web dasboard to visualise the queries. It is based on a Node.js server, Socket.io for realtime data exchange and Angular 2.0 on the client side.

Requirements:

- Node.js ¿= 6.0.0
- AngularJS 2.0
- or Docker >= 1.10

[†] https://github.com/Murthy10/storm
[‡] https://github.com/Murthy10/StormBoard