Big Data analytics on high velocity streams

Specific use cases with Storm

Master Thesis

Thibaud Chardonnens

June 2013

Thesis supervisors:

Prof. Dr. Philippe Cudre-Mauroux
eXascale Information Lab
Benoit Perroud
VeriSign Inc.

UNIVERSITÉ DE FRIBOURG SUISSE
UNIVERSITÄT FREIBURG SCHWEIZ

Software Engineering Group
Department of Informatics
University of Fribourg (Switzerland)

"It is a capital mistake to theorize before one has data."

- *Sherlock Holmes*

Acknowledgements

The last two years, while I was doing my Master in Information Management at the University of Fribourg, was probably the most intense and meaningful period for me. I learnt from a multitude of exciting topics that fascinated me. Therefore, I would like to thank all my professors who gave me their passion.

In the scope of this thesis, I would especially like to thank Philippe Cudre-Mauroux who was the supervisor of this work. He and all his team were really present and helpful to me. They gave me a lot of guidance and helped me to set up the cluster of machines used for the use cases.

This thesis is done in partnership with VeriSign Inc. Benoît Perroud was the person of contact from VeriSign. He was really helpful and gave a lot of wise and meaningful advices. I learnt a lot due to this collaboration

Last but not least, I also want to thank Cyrielle Verdon who supported me during my studies. She helped me to write and correct this thesis and my English, which is far for being perfect.

Abstract

The presented research work provides a complete solution for processing high velocity streams in real-time. For the processing part, we mainly used **Storm**, but other tools will also be analyzed in this thesis.

Storm is a real-time distributed computation system that is fault-tolerant and guarantees data processing.

In order to provide all the notions for understanding the subject, this thesis also describes the following topics :

- Big Data: very trendy topic that represents the huge amount of data created during the last few years
- NoSQL: term used to designate database management systems that differ from classic relational database management system (RDBMS) in some way [Per13a]
- Hadoop: nowadays the most common tool for processing huge amount of data in a distributed manner

The architecture proposed in this thesis allows to process both Twitter and Bitly streams and extract insights in real-time. Storm was used as computation system but, in order to provide a complete solution, the following tools will also be used and analyzed:

- Cassandra: a NoSQL database good for both reads and writes
- Redis: an advanced key-value store
- Kafka: a high-throughput queue messaging system

Keywords: Storm, Real-Time, Big Data, Stream processing, NoSQL, Cassandra, Kafka, Hadoop, MapReduce, Java

Table of Contents

1. Introduction **2**

1.1. Motivation and Goals . . . 2
1.2. Organization . . . 3
1.3. Notations and Conventions . . . 4

I. Big Data Background **5**

2. Digging into Big Data **6**

2.1. What is Big Data? . . . 6
2.2. How big are data? . . . 9
2.2.1. Globally . . . 9
2.2.2. By sectors . . . 10
2.3. Where does that data come from? . . . 11
2.4. The 3-Vs of Big Data . . . 12

3. NoSQL databases **13**

3.1. NoSQL databases . . . 13
3.2. Scaling horizontally with a traditional database . . . 14
3.3. The CAP theorem . . . 16

4. Cassandra **19**

4.1. Introduction to Cassandra . . . 19
4.2. Architecture . . . 20
4.2.1. Architecture overview . . . 20
4.2.2. Internode communication . . . 20
4.2.3. Data distribution and replication . . . 20
4.3. Data Model . . . 21
4.4. Query Model . . . 24

5. Hadoop **26**
5.1. Introduction to Hadoop 26
5.2. Hadoop File System (HDFS) 27
5.3. MapReduce 28
5.4. Hadoop limitations 29

II. Real time data processing and Storm **31**

6. Real time data processing and Big Data **32**
6.1. Real time data processing 32
6.2. Real-time data processing systems 34
6.2.1. MapReduce Online 35
6.2.2. Apache Drill 35
6.2.3. Cloudera Impala 37
6.2.4. StreamBase 37
6.2.5. IBM InfoSphere 37
6.2.6. Truviso 38
6.2.7. Yahoo S4 38
6.2.8. Storm 38
6.3. The Lambda architecture 39

7. Introduction to Storm **41**
7.1. What is Storm? 41
7.2. Components of a Storm cluster 42
7.3. Streams and Topologies 42
7.4. Parallelism hint 44
7.5. Stream groupings 45
7.6. Fault-tolerance in Storm 45
7.7. Guaranteeing message processing 47
7.7.1. Transactional topologies 50
7.8. Trident 50
7.9. Storm UI 51

III. Storm : use cases **52**

8. Input data **53**
8.1. Twitter 53
8.1.1. Public streams 54
8.1.2. Data format 54

8.1.3. Rate limit 55
8.1.4. Other providers of Twitter data 55
8.2. Bitly 55
8.3. Link between Twitter and Bitly 57

9. Bitly Topology **59**
9.1. Description of this use case 59
9.2. Spout implementation 62
9.3. Values extraction and blacklist check 63
9.4. SLD/TLD count and get webpage title 63

10. Twitly topology **67**
10.1. Description 67
10.2. Pushing data into Kafka 68
10.3. Data processing and storage 70
10.4. Trends detection 71
10.5. Results 73

11. Performance and Storm Evaluation **76**
11.1. Method employed 76
11.2. Results 77
11.3. Storm evaluation 78

12. Conclusion **80**

A. Common Acronyms **81**

References **83**

Index **86**

List of Figures

2.1. The interest for the term "Big Data" on Google (december 2012) 7
2.2. The Hype Cycle of emerging technologies. 8
2.3. Global storage in exabytes 10
2.4. Repartition of data by sector in US 11
2.5. The repartition between analog and digital data from 1986 to 2007 . . . 12

3.1. Relational schema for the analytics application 14
3.2. Visual Guide to recent Systems [Hur] 18

4.1. Relational VS Cassandra Data Model 22
4.2. Composite Column 24

5.1. Hadoop basic architecture 27
5.2. HDFS architecture [Fouc] 28
5.3. MapReduce word count example 29
5.4. MapReduce is limited to batch processing 30

6.1. Real-time : processing the latest version of the data 33
6.2. Real-time : getting response of query in real-time 34
6.3. Overview of Drill's architecture 36
6.4. The lambda architecture 39

7.1. An example of a simple Storm Topology 43
7.2. Relationships of worker processes, executors and tasks. [Nol12] 44
7.3. Tree of tuples 48

9.1. Schema of the Bitly Topology 62
9.2. Bloomfilter: false positive 65

10.1. Usage of the machines 69
10.2. Overview of the Twitly use case 70

10.3. Sliding Windows [Nol13] 72
10.4. Trends detection with a rolling count algorithm 73
10.5. Data visualization with HighCharts 74
10.6. Data visualization with Timeline JS 75

11.1. Scalability 77

List of Tables

4.1. Hash on primary key 21
4.2. Data distribution based on Hash 21
4.3. Data Types in Cassandra 23

7.1. Built-in Stream grouping [Marb] 46

8.1. Parameters for the "POST statuses/filter" endpoint 54

11.1. SWOT analysis of Storm 79

Listings

4.1. Creating a keyspace 22
4.2. Creating a Column Family 22
4.3. Insert a single column using Hector 25

8.1. A Tweet encoded in JSON 56
8.2. A Bitly tuple in JSON 57

9.1. The main method of a Storm Topology 60
9.2. Spout's implementation 64

1 Introduction

Over the last few years, data stored in the world has increased exponentially. Data comes from everywhere: companies store information about their stakeholders and they use more and more machines that log what they are doing, sensors store information about their environment, people can share information on social platforms from everywhere with a smartphone, etc. This phenomenon is called "*Big Data*" and is a very trendy topic nowadays.

For processing this huge amount of data, new tools and approaches are required. Currently, the most used tool is definitely *Hadoop*. Even if Hadoop does its job very well, it is limited to process data by batch and is certainly not a good fit for processing the latest version of data.

More and more cases require to process data almost in real-time, which means as data arrive, in order to react and take decisions quickly after an event happened. This kind of computation is generally called *stream processing*, in other words processing a constant flux of data, in real-time.

In stream processing, rather than the notion of volume, the notion of velocity is determinant. The speed at which new data is created and needs to be process can be extremely high. Therefore, new tools for processing this huge velocity of data are required.

One new tool is called *Storm*, it becomes more and more popular and seems to be really promising. Storm does for realtime processing what Hadoop did for batch processing. Consequently, Storm will be deeply analyzed and used in this thesis.

1.1. Motivation and Goals

Nowadays, processing big amount of data is quite common thanks to Hadoop and MapReduce. However, processing internet-scale amount of data with low latency update, almost in real-time, is really challenging for the near future. In more and more use cases it is important to know what is happening now and take decision as quickly as possible.

Therefore, the main goals of this thesis are:

- Provide the basic notions about Big Data;
- Explore the state of the art in real-time data stream processing
- Analyze Storm software and identify its strengths and weaknesses
- Implement specific use cases provided by the industrial partner VeriSign by developing a Storm-based prototype system

1.2. Organization

This thesis is divided into three main sections :

I **Big Data Background**: this section focuses on providing several concepts about the notion of Big Data:

- Some definitions and numbers in chapter 2
- An overview of NoSQL databases and the CAP theorem in chapter 3
- A description of Cassandra, a NoSQL database used in our use case in chapter 4
- An overview of Hadoop, nowadays the most common tool for processing huge amount of data in chapter 5.

II **Real-time data processing and Storm**: this section describes the notion of real-time and tools for processing data with low latency. Storm will also be described in detail in this section:

- A description of what is real-time and tools for processing stream of data in chapter 6
- An introduction to Storm in chapter 7

III **Storm: use cases**: This last section will present the implementation done with Storm:

- First, the input data used will be described in chapter 8
- Chapter 9 explains the first use case done
- Chapter 10 will focus on the second use case, which mixes both Twitter and Bitly data
- The performance of the use cases and an evaluation of Storm will be done in chapter 11

1.3. Notations and Conventions

- Formatting conventions:
  - **Bold** and *italic* are used for emphasis and to signify the first use of a term.
  - SansSerif is used for web addresses.
  - Code is used in all Java code and generally for anything that would be typed literally when programming, including keywords, constants, method names, and variables, class names, and interface names.
- The present report is divided in Chapters. Chapters are broken down into Sections. Where necessary, sections are further broken down into Subsections, and Subsections may contain some Paragraphs.
- Figures, Tables and Listings are numbered inside a chapter. For example, a reference to Figure j of Chapter i will be noted *Figure i.j*.
- As far as gender is concerned, I systematically select the feminine when possible.
- Source code is desplayed as follows:

```
1 Matrix3f rotMat = new Matrix3f();
  rotMat.fromAngleNormalAxis( FastMath.DEG_TO_RAD * 45.0f, new Vector3f( 1.0f, 0.0f, 0.0f));
3 box.setLocalRotation( rotMat );
```

Part I.

Big Data Background

2

Digging into Big Data

2.1. What is Big Data? **6**

2.2. How big are data? **9**

2.2.1. Globally 9

2.2.2. By sectors 10

2.3. Where does that data come from? **11**

2.4. The 3-Vs of Big Data **12**

This chapter describes the challenge of Big Data Management from both economic and informatics points of view.

*

"Welcome to the Age of Big Data. The new megarich of Silicon Valley, first at Google and now Facebook, are masters at harnessing the data of the Web - online searches, posts and messages - with Internet advertising. At the World Economic Forum last month in Davos, Switzerland, Big Data was a marquee topic. A report by the forum, "Big Data, Big Impact," declared data a new class of economic asset, like currency or gold." [Loh12]

*

2.1. What is Big Data?

There is not a perfect definition of Big Data. The term is used by many companies and becomes more and more popular as shown in Figure 2.1 with Google's tool: Google Trend. To complete the Figure 2.1 we can also have a look at the famous Gartner's Hype Cycle of emerging technologies (Figure 2.2). In july 2012, according to Gartner, Big Data leaves the "Technology Trigger" phase and enter in the "peak of inflated expectations"

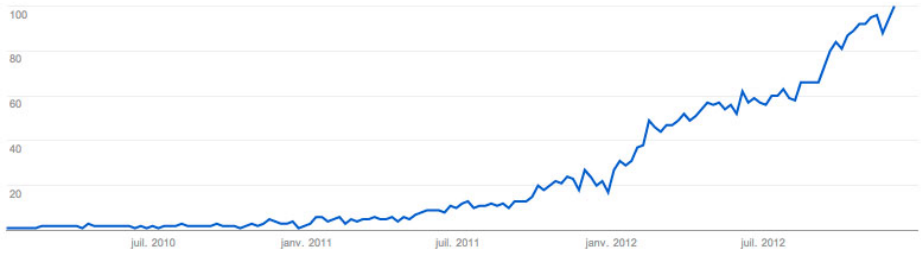


Figure 2.1.: The interest for the term "Big Data" on Google (december 2012)

phase. Without any doubt, Big Data will be a very popular term in the next years. At first glance, Big Data refers to something large and full of information but let's precise this concept.

In some ways, we are all dealing with (big) data. Users create content like blog posts, tweets, photos and videos on social networks. Servers continuously log messages about what they're doing. Scientists create detailed measurements, companies record information about sales, suppliers, operations, customers, etc. In 2010, more than 4 billion people (60 percent of the world's population) were using mobile phones, and about 12 percent of those people had smartphones, whose penetration is growing more than 20 percent a year. More than 30 million networked sensor nodes are now present in the transportation, automotive, industrial, utilities and retail sectors. The number of these sensors is increasing at a rate of 30 percent a year. The ultimate source of data, Internet, is therefore incomprehensibly large. A study shows that every year the amount of data will grow by 40 percent to 2020 [Ins11].

This explosion of data has profoundly affected businesses. Traditional database systems, like relational databases (RDBMS), show their limits. Traditional systems have failed to scale to big data. One could wonder why? The problem is simple : while the storage capacity of hard drives have increased massively over the years, access speed (the rate at which data can be read from drive) is unable to keep up. In 1990, one typical drive could store about 1000 MB of data and had a transfer speed of 4.4MB/s[1]. Therefore it was possible to read the whole disk in around five minutes. Over 20 years later, one terabyte drives are the norm but the transfer speed is around 100MB/s. Therefore, it takes more than two and a half hours to read all the data off the disk (and writing is even slower). The obvious way to reduce the time is to read from multiple disks at once [Whi12].

So a good definition of "Big data" may be that from a paper by the McKinsey Global Institute in May 2011: "Big Data refers to data sets who size is beyond the ability of typical database software tools to capture, store, manage and analyze" [Ins11].

[1]These specifications are for the Seagate ST-41600n.

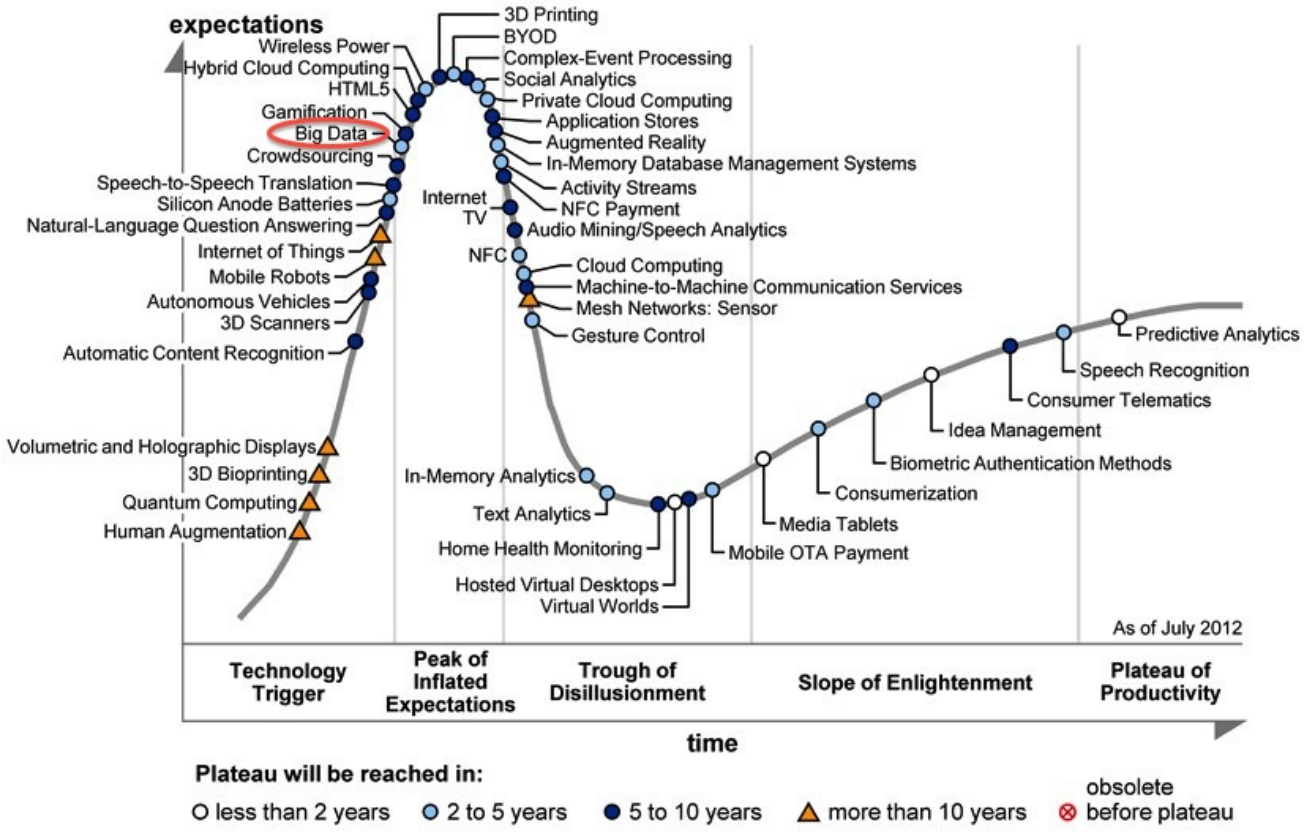


Figure 2.2.: The Hype Cycle of emerging technologies.

To tackle the challenges of Big Data, a new type of technologies has emerged. Most of these technologies are distributed across many machines and have been grouped under the term "NoSQL". NoSQL is actually an acronym that expands to "Not Only SQL". In some ways, these new technologies are more complex than traditional databases, and in other ways they are simpler. There isn't any solution that fits all situations. You must do some compromises. This point will also be analyzed in this thesis. These new systems can scale to vastly larger sets of data, but using these systems require also new sets of technologies. Many of these technologies were first pioneered by two big companies: Google and Amazon. The most popular is probably the MapReduce computation framework introduced by Google in 2004. Amazon created an innovative distributed key-value store called Dynamo. The open source community responded in the year following with Hadoop (free implementation of MapReduce), HBase, MongoDB, Cassandra, RabbitMQ and countless other projects [Mar12].

Big Data is a big challenge, not only for software engineers but also for governments and economists across every sectors. There is a strong evidence that Big Data can play a significant role not only for the benefit of private commerce but also for that of national economies and their citizens. Studies from McKinsey Global Institute show, for example, that if US health care could use big data creatively and effectively to drive efficiency

and quality, the potential value from data in the sector could reach more than $300 billion every year. Another example shows that, in the private sector, a retailer using big data to the full has the potential to increase its operating margin by more than 60 percent [Ins11]. Therefore, another interesting definition of Big Data, more economical, is that from IDC : "Big data technologies describe a new generation of technologies and architectures, designed to economically extract value from very large volumes of a wide variety of data, by enabling high-velocity capture, discovery, and/or analysis" [IDC11].

2.2. How big are data?

This section will try to give an idea of the growth of data in recent years. Big Data has reached a critical mass in every sector and function of the economy. The rapid development and diffusion of digital information technologies have, of course, intensified its growth [Ins11].

This section is divided into two subsections; firstly, we will analyze the amount of data around the whole world and, then, more locally in different sectors.

2.2.1. Globally

To start, let us go back a few years, at the start of the decade. Hal Varian and Peter Lyman from University of California Berkeley, were pioneers in the research on the amount of data produced, stored and transmitted. For their "How much Information?" project that ran from 2000 to 2003, they estimated that 5 exabytes (as a reminder 1 exabyte = 1 000 000 terabytes) of new data were stored globally in 2002, and that, more than three times this amount, 18 exabytes, of new or original data were transmitted, but not necessarily stored, through electronic channels such as radio, telephone, television and internet. They estimated that the amount of new data stored doubled from 1999 to 2002, a compound annual growth rate of 25 percent [PL03].

A few years later, in 2007, the information-management company EMC sponsored the research firm IDC [IDC11] to produce an annual serie of reports on the "Digital Universe" to size the amount of data created and replicated each year. This study showed that, for the first time in 2007, the amount of digital data created in a year exceeded the world's data storage capacity. In other words, it was impossible to store all the data being created, because the rate at which data generation is increasing is much faster than the expansion of world's data storage capacity. But reminder that a gigabyte of stored content can generate a petabyte of transient data that we typically don't store.

The study of IDC estimated that, in 2009, the total amount of data created and replicated was 800 exabytes, and surpasses 1.8 zettabytes in 2011, growing by a factor of 9 in just

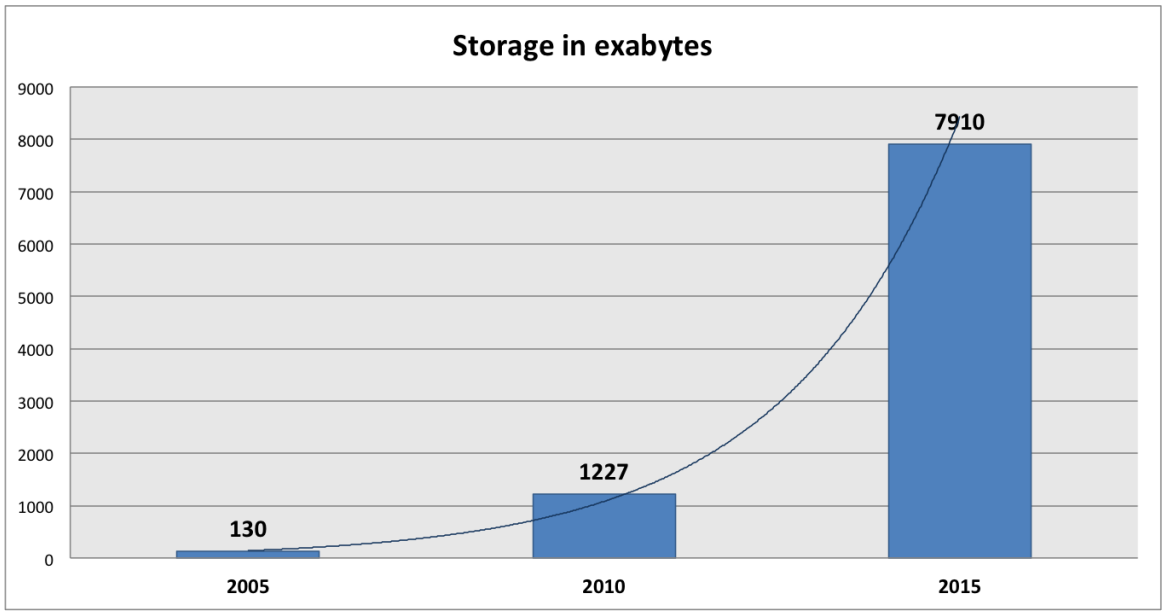


Figure 2.3.: Global storage in exabytes

five years. They projected that the volume would grow by more than 40 times to 2020, an implied annual growth rate of 40 percent[2]. Figure 2.3 represents this explosion of data through a graph.

2.2.2. By sectors

There is a wide agreement that data creation is growing exponentially, but is this growth concentrated only in specific segments of the global economy? The answer is no. The growth of Big Data is a phenomenon that is observed in every sector.

McKinsey Global Institute estimated that, by 2009, almost all sectors in the US economy had an average of 200 terabytes of stored data per company with more than 1000 employees. Many sectors had even more than one petabyte in mean stored data per company. And of course, some individual companies have much higher stored data than the average of their sector, and potentially more opportunities to capture value from big data[Ins11].

Figure 2.4 shows that financial services sectors, including investment services, securities and banking, have on average the most digital data per firm (for example, the New York Stock Exchange, boasts about half a trillion trades a month). Government, communications and media firms and utilities also have a significant amount of data stored per enterprise or organization. Discrete and process manufacturing have the highest aggre-

[2]IDC estimates of the volume of data include copies of data, not just originally generated data

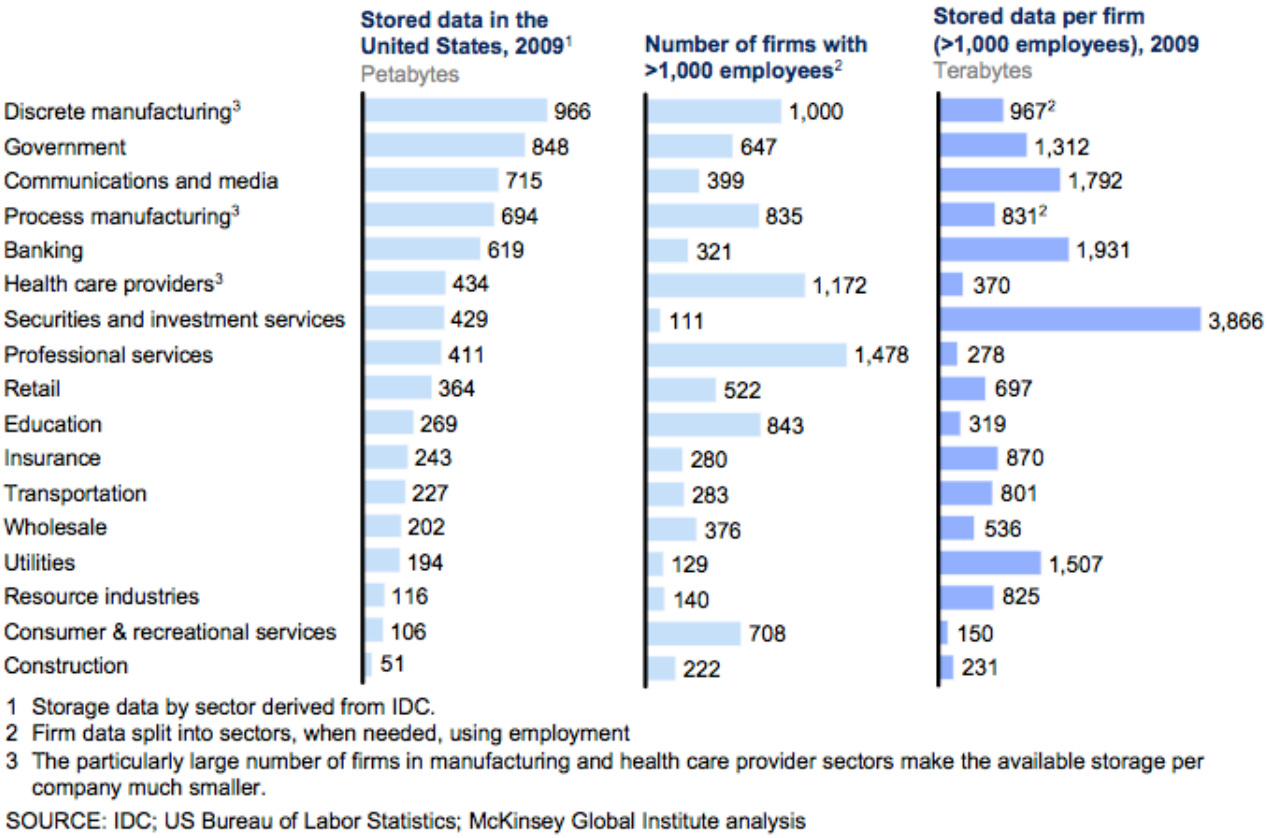


Figure 2.4.: Repartition of data by sector in US

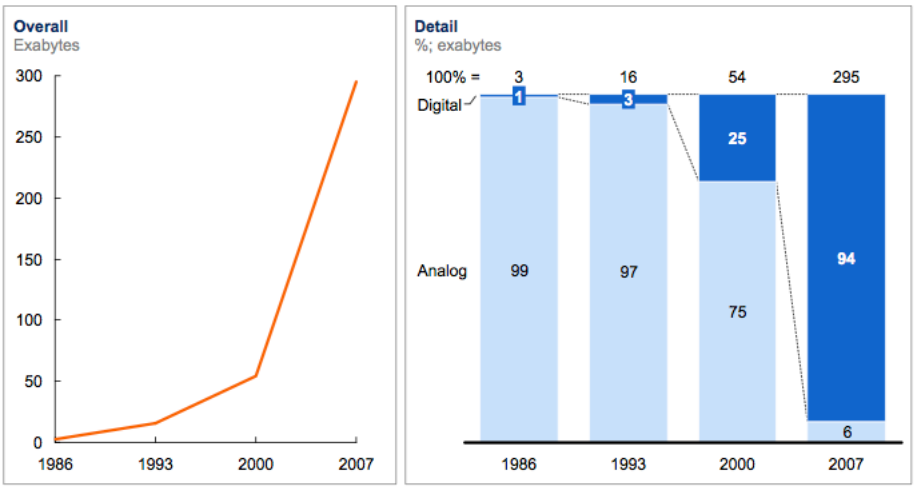
gate amount of data stored. However, these sectors are fragmented into a large number of firms. Therefore they rank much lower in intensity terms.

This data comes from the US and there isn't such a study for Europe. However, we can assume that the numbers would be quite similar.

2.3. Where does that data come from?

The volume of data has really started to explode with the transition from analog to digital. As shown in Figure 2.5 , in 1986 digital data represented only one percent of data. The biggest change came in the years following 2000; in this period digital data rose from 25% to reach 94% in 2007 [Ins11]. Since, several other factors contribute each day to this expansion :

- Technological developments
- Social platforms
- Fusion and acquisition
- Predominance of systems ERP and CRM



NOTE: Numbers may not sum due to rounding.
SOURCE: Hilbert and López, "The world's technological capacity to store, communicate, and compute information," *Science*, 2011

Figure 2.5.: The repartition between analog and digital data from 1986 to 2007

- Multiplication of machines

2.4. The 3-Vs of Big Data

As seen in the last sections, Big Data represents large sets of data, but the notion of volume isn't the only one to be considered. A high velocity and variety are generally also associated with Big Data. **Volume**, **velocity** and **variety** are called the 3-Vs of Big Data.

The velocity describes the frequency at which data are generated and caught. Due to recent technological developments, consumers and companies generate more data in shorter time. The problem with an high velocity is the following : nowadays, the most widely used tool for processing large data sets is Hadoop and more precisely the MapReduce framework. Hadoop processes data by batch, so when the process started, new incoming data won't be taking into account in this batch but only in the next one. The problem is even bigger if the velocity is high and if the batch processing time is long. These questions will be discussed in chapter 5.

Nowadays, companies aren't only dealing with their own data. More and more data necessary to perfectly understand the market are generated by third parties outside the company. Therefore, data comes from many different sources in various types, it could be text from social network, image data, geo location, logs, sensors data etc. The wide variety of data is a good example of why traditional databases,that are designed to accept always the same data structure, are not a good fit for Big Data.

3

NoSQL databases

3.1. NoSQL databases 13
3.2. Scaling horizontally with a traditional database 14
3.3. The CAP theorem 16

In the previous chapter, the growth of data was discussed. In this chapter, we will focus on how big amount of data can and should be stored.

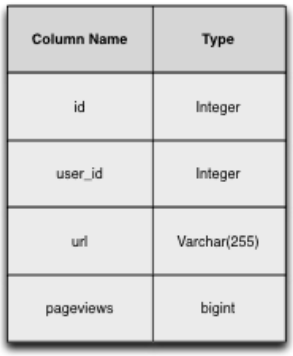
3.1. NoSQL databases

For people who have strong experiences in relational databases, it can be really disturbing and confusing to understand some principles of non-relational databases. As said in the previous chapter, NoSQL systems can be more complex than traditional databases, and in other ways they are simpler. These systems can scale to vastly larger set of data but in return we some compromises must be done. With NoSQL databases the data model is generally much more limited than what we are used to with something like pure SQL.

NoSQL (Not only SQL) is a term used to designate database management systems that differ from classic relational database management system (RDBMS) in some way. These data stores may not require fixed table schemas, usually avoid join operations, do not attempt to provide ACID properties and typically scale horizontally [Per13a].

An important aspect of the previous definition is the notion of scalability. Scalability can be described as a desirable property of a system, a network, or a process, which indicates its ability to either handle growing amount of work in a graceful manner or to be readily enlarged [Per13a]. Scalability can be done in two dimensions :

- **Scale up** : scale vertically (increase RAM in an existing node)
- **Scale out** : scale horizontally (add a node to the cluster)



Column Name	Type
id	Integer
user_id	Integer
url	Varchar(255)
pageviews	bigint

Figure 3.1.: Relational schema for the analytics application

NoSQL databases generally scale horizontally; if the database needs more resources (storage, computational power, memory, etc.), one or more nodes should be easily added to the cluster. However, adding more machines increases the risk of failures (network issues, insufficient disk space or memory, hardware issues, etc.). Therefore, an essential property of a distributed system is fault-tolerance. Fault-tolerance is the property that enables a system to continue operating properly in the event of failure of some of its components [Wik13b].

NoSQL databases are very different in their data model but also in their architecture. Does it mean that relational databases should no longer be used? Of course not, it's not possible to have one size fits all solutions. RDBMS are very powerful with reasonable amount of data and good for joins. Furthermore the data model of traditional databases is easy to understand, very well known and strong. Therefore, it would be very interesting to ask ourselves what could concretely happen if we try to deal with a huge amount of data with a classic RDBMS. This is what will be discussed in the next section.

3.2. Scaling horizontally with a traditional database

Let's assume that we want to build a new web analytics application. The application simply tracks the number of pageviews to any URL a customer wishes to track. The traditional relational schema for the page views looks like Figure 3.1. Everytime an URL tracked is viewed, the webpage pings your application and your server increments the corresponding row in the RDBMS [Mar12].

This schema makes sense, but now let's imagine that our application is a great success and the traffic is growing exponentially. We will soon discover a lot of errors saying all something like : "Timeout error on writing to the database". The database cannot keep up with the load. Therefore, write requests to increment page views are timing out.

For fixing this problem we have two main choices : scaling our system with a queue and/or by sharding the database.

- **Scaling with a queue** : Doing only a single increment at a time to the database is wasteful, it would be much more efficient if we batch many increments in a single request. So, instead of having the web server hitting the database directly, you insert a queue and a worker process between the web server and the database. Whenever we receive a new page view, that event is added to the queue and the worker process reads, for example, 1000 events at the time off the queue and batch them into a single database update [Mar12].
- **Scaling by sharding the database** : Let's assume that our application continues to get more and more popular. Scaling the database with a queue is only a temporary solution, our worker cannot keep up with the writes even if we try to add more worker to parallelize the updates; the database is clearly the bottleneck.

  We need to spread our data across many servers. Each server will have a subset of the data for the table. This is known as "horizontal partitioning" or "sharding". In order to choose a shard for each key, we need to take the hash of the key modded by the number of shards. Then we need to write a script to map over all the rows in our single database instance and split the data into for example four shards. Finally, all of our application code needs to know how to find the shard for each key. So we wrap a library around our database handling code that reads the number of shards from a configuration file and redeploys all of our application code. [Mar12]

Even if these solutions are feasible, they are far from being easy and perfect. Some problems could occur [Mar12]:

1. **Scaling is hard**. As the application becomes more and more popular, we keep having to reshard the database into more shards to keep up with the write load. Each time gets more and more painful as there's much more work to coordinate.
2. **Fault-tolerance issues** : The more machines we have, the more chances we have that one of them goes down. If it happens (and it will happen), the portion of data stored by the machine that goes down is unavailable and if we didn't setup our system correctly, the whole system will be unavailable.
3. **Corruption / human fault-tolerance issues** : Imagine that while working on the code we accidentally deploy a bug to production that increments the number of page view by two for every URL instead of one. Many of the data in our database will be inaccurate and there's no way of knowing which data got corrupted.
4. **Complexity pushed to application layer** : Our application needs to know which shard to look at for every key. For a query like "Top 100 URLs" you need to query every shard and then merge the results together.

5. **Maintenance is an enormous amount of work** : Scaling a sharded database is time-consuming and error-prone. We have to manage all the constraints. What we really want is for the database to be self-aware of its distributed nature and manage the sharding process for us.

As we can see, trying to distribute a traditional database isn't impossible but generally really painful. With NoSQL databases the problems of sharding, fault-tolerance, maintenance and complexity are hidden because this kind of databases are generally built to be distributed.

However managing Big Data is always hard, strictly speaking it's actually **impossible**. As seen, for storing Big Data we need distributed systems because one machine is generally not enough and if a part of the system is down the whole system should continue to operate. Of course we want these systems to be always available and consistent. As we will see in the next sub-section it's impossible to get these three properties (partition-tolerance, availability, consistency) together.

3.3. The CAP theorem

The theorem began as a conjecture made by University of California, Berkeley computer scientist Eric Brewer at the 2000 Symposium on Principles of Distributed Computing (PODC). In 2002, Nancy Lynch and Seth Gilbert of MIT published a formal proof of Brewer's conjecture, rendering it a theorem[Wik13a].

The CAP theorem states that it is impossible for a system to provide simultaneously all the three following properties :

- **Consistency** (all nodes see the same data at the same time)
- **Availability** (a guarantee that every request receives a response)
- **Partition-Tolerance** (No set of failures less than total network failure is allowed to cause the system to respond incorrectly)

All of these properties are desirable and expected. But depending on our needs we can pick at most two of them. We can therefore have systems that provide consistency and availability (CA), consistency and partition-tolerance (CP) and finally availability and partition-tolerance (AP). Let's define in more detail these three properties.

Nowadays, most databases attempt to provide strongly *consistent* data. Most of the new frameworks for designing distributed services depend on ACID databases. ACID is an acronym for Atomic, Consistent, Isolated and Durable. Interactions with such databases should behave in a transactional manner : "operations commit or fail in their entirety (atomic), committed transactions are visible to all future transactions (consistent), uncommitted transactions are isolated from each other (isolated), and once a transaction is

committed it is permanent (durable)" [MIT02]. For example, it is essential that billing information and commercial transaction records are handled with this type of strong consistency.

Databases are also expected to be highly *available*. It means that every request should succeed and receive a response in a reasonable delay. The classic example of issues caused by a down database is the uncertainty about if our commercial transaction has been completed or not. This problem is exacerbated by the fact that a database is most likely to be unavailable when it is most needed.[MIT02]

Finally, on a distributed network, it is essential to provide *fault-tolerance*. There are almost an unbounded list of problems that may happen on a cluster of nodes: insufficient disk space or memory, hardware issues, network failures etc. So when some nodes crash, it is important that the whole service still performed as expected. Gilbert and Lynch define partition tolerance as : "No set of failures less than total network failure is allowed to cause the system to respond incorrectly."[MIT02]

CAP theorem comes to life as an application scales : at low transaction volumes, small latencies to allow databases to get consistent have no noticeable affect on either the user experience or the overall performance. But as activity increases, these few milliseconds in throughput will grow and possibly create errors. For example, Amazon claims that just an extra one tenth of second on their response times will cost them 1% in sales. Google said they noticed that just a half a second increase in latency caused traffic drop by fifth. The problem becomes even more significant with errors; imagine that we enter our credit card number and just after validating the transaction we get an error. We wonder whether we have just paid for something we won't get, not paid at all, or maybe the error is immaterial to this transaction ...[Bro]

In a distributed system, it is not possible to avoid network partitions. Therefore the right balance between consistency and availability must be find. To illustrate this, let's consider two big platforms : Facebook and Ebay. If a status update is done on Facebook it isn't really a problem if it is not updated at the same time all around the world. Users prefer to have a platform which is always available than fully consistent. Facebook should, therefore, use system that provide partition-tolerance and availability (AP). Consequently Facebook made Cassandra, an eventual consistent database. On the other hand, let's imagine that someone want to buy a product on Ebay and there is only one copy in stock. Just before paying, the website shows an error saying that there is actually no more product available. For this precise case it's maybe not a huge issue, but at the scale of thousands of inconsistencies, the problem become real and big. Therefore, a platform like Ebay should use a system that provide consistency and partition-tolerance (CP).

Figure 3.2 provides a small classification (subject to discussion) of actual systems. Systems that provide consistency and availability have problems with partitions and typically deal with it with replication. Examples of CA systems include traditional RDBMSs

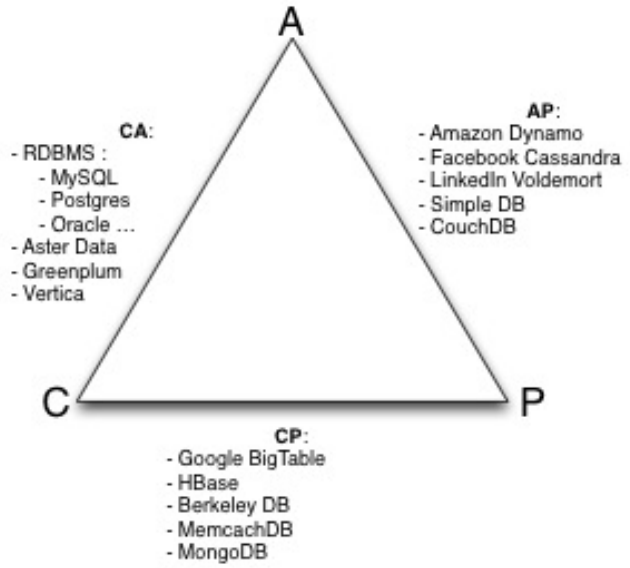


Figure 3.2.: Visual Guide to recent Systems [Hur]

like MySQL or Postgres, Vertica (column-oriented), Aster Data (relational) and Greenplum (relational). Cassandra (column-oriented/tabular), Amazon Dynamo (key-value) or CouchDB (document-oriented) are systems that are eventually consistent, available and tolerant to partitions (AP). Finally, systems like Google BigTable (column-oriented/tabular), Hbase (column-oriented/tabular), MongoDB (document-oriented) and BerkeleyDB (key-value) are consistent and tolerant to partitions (CP) but can have trouble with availability.

4

Cassandra

4.1. Introduction to Cassandra 19

4.2. Architecture 20

4.2.1. Architecture overview 20

4.2.2. Internode communication 20

4.2.3. Data distribution and replication 20

4.3. Data Model 21

4.4. Query Model 24

This chapter focuses on Cassandra, a very well known NoSQL database. Furthermore Cassandra is the database that we will use in our use cases with Storm in the third part of the report.

4.1. Introduction to Cassandra

Cassandra is an open source distributed database management system. It is a NoSQL solution initially developed at Facebook and part of the Apache Software Foundation since 2008. Cassandra is designed to handle large amounts of data spread out across many nodes, while providing a highly available service with no single point of failure. In term of CAP theorem, Cassandra provides availability, partition-tolerance (AP) and eventual consistency (or tunable consistency).

Unlike RDBMS, Cassandra does not support a full relational data model, it could be simplified as a scalable, distributed, sparse and eventually consistent hash map [Per13a].

4.2. Architecture

4.2.1. Architecture overview

"Cassandra is designed to handle big data workloads across multiple nodes with no single point of failure. Its architecture is based in the understanding that system and hardware failure can and do occur. Cassandra addresses the problem of failures by employing a peer-to-peer distributed system where all nodes are the same and data is distributed among all nodes in the cluster. Each node exchanges information across the cluster every second. A commit log on each node captures write activity to ensure data durability. Data is also written to an in-memory structure, called a memtable, and then written to a data file called an SStable on disk once the memory structure is full. All writes are automatically partitioned and replicated throughout the cluster." [Data]

Any authorized user can connect to any node in any datacenter and access data using the CQL (Cassandra Query Language) or the CLI (Command Line Interface). "Client read or write requests can go to any node in the cluster. When a client connects to a node with a request, that node serves as the coordinator for that particular client operation. The coordinator acts as a proxy between the client application and the nodes that own the data being requested. The coordinator determines which nodes in the ring should get the request based on how the cluster is configured." [Data]

4.2.2. Internode communication

For internode communication, Cassandra uses a protocol named *gossip* to discover location and state information about other nodes in the cluster. Gossip is a peer-to-peer communication protocol in which nodes exchange state information about themselves and about other nodes they know about.

The gossip process runs every second and exchanges state information with a maximum of three other nodes. The nodes exchange information about themselves and also about other nodes that they have gossiped about. In this way, all nodes quickly learn about all other nodes in the cluster. Each gossip message contains a version, so older information are overwritten with the most current state. [Data]

4.2.3. Data distribution and replication

Cassandra is designed as a peer-to-peer system that makes copies of the data and replicates the data among a group of nodes. Data is organized by tables and each row is identified by a primary key. For distributing the data across the cluster, Cassandra uses a consistent hashing mechanism based on the primary key. Therefore the primary key

Name	Murmur3 hash value
Jim	-2245462676723223822
Carol	7723358927203680754
Johny	-6723372854036780875
Suzy	1168604627387940318

Table 4.1.: Hash on primary key

Node	Murmur3 start range	Murmur3 end range
A	-9223372036854775808	-4611686018427387903
B	-4611686018427387904	-1
C	0	4611686018427387903
D	4611686018427387904	9223372036854775807

Table 4.2.: Data distribution based on Hash

determines which node the data is stored on. Copies of rows are called replicas. When data is first written, it is also referred to as a replica [Data].

When creating a cluster, it is important to specify the replication strategy which determines the replicas for each row of data and the snitch which defines the topology information that the replication strategy uses to place replicas.

The Table 4.1 shows the relation between the primary key and its hash value (created with Murmur 3).

Each node in the cluster is responsible for a range of data based on the hash value. For example if the cluster is composed of four nodes the distribution can be represented like the Table 4.2.

Then Cassandra places the data on each node according to the value of the hash on the primary key and the range that the node is responsible for. For example, the row that have the primary key "Johny" will be stored on the node A.

4.3. Data Model

The Cassandra data model is a *dynamic schema*, *column-oriented* data model. Unlike a relational database, it is not necessary to model all of the columns required by the application, each row is not required to have the same set of columns. The different components of the data model can be represented like the following :

$$\boxed{\text{Keyspace}} > \boxed{\text{Column Family}} > \boxed{\text{Row}} > \boxed{\text{Column Name}} = \boxed{\text{Value}}$$

Figure 4.1 compares the SQL model and the Cassandra data model.

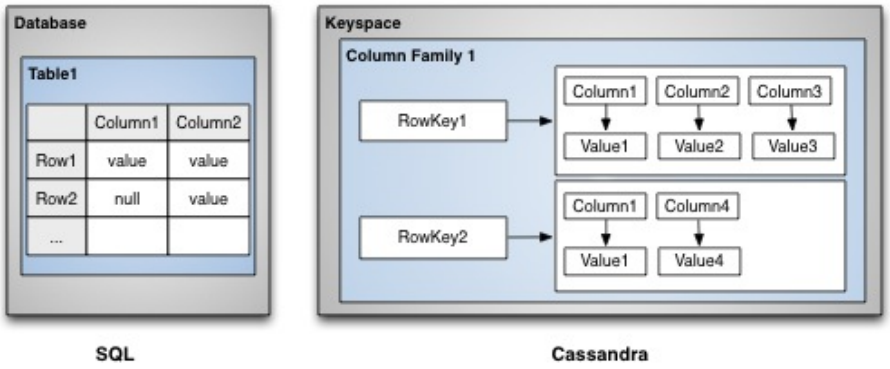


Figure 4.1.: Relational VS Cassandra Data Model

A Keyspace can be compared to a database in SQL words. The Keyspace defines the replication factor and the network topology. The Listing 4.1 describes how to create a new keyspace with the command line interface (CLI).

```
CREATE KEYSPACE demo
  WITH placement_strategy = 'org.apache.cassandra.locator.SimpleStrategy'
  AND strategy_options = {replication_factor:1};
```

Listing 4.1: Creating a keyspace

A Column Family is the equivalent to an SQL table. When creating a new column family it is recommended to specify the comparator and the validator. The validator represents the data type for a column value and the comparator represents the data type for a column name. The Table 4.3 lists all the data types in Cassandra and the Listing 4.2 shows how to create a new column family with information about column metadata with the command line interface (CLI).

```
CREATE COLUMN FAMILY users
  WITH comparator = UTF8Type
  AND key_validation_class=UTF8Type
  AND column_metadata = [
    {column_name: full_name, validation_class: UTF8Type}
    {column_name: email, validation_class: UTF8Type}
    {column_name: state, validation_class: UTF8Type}
    {column_name: gender, validation_class: UTF8Type}
    {column_name: birth_year, validation_class: LongType}
  ];
```

Listing 4.2: Creating a Column Family

A column family contains rows. A row is defined by a unique key. There are two strategies of key partitioner on the key space. A partitioner determines how data is distributed across the cluster. A partitioner is a hash function for computing the token of a row key. The two strategies are :

- RandomPartitioner : Uniformly distributes data across the cluster based on Murmurhash or md5 hash values.

Internal Type	CQL Name	Description
BytesType	blob	Arbitrary hexadecimal bytes (no validation)
AsciiType	ascii	US-ASCII character string
UTF8Type	text, varchar	UTF-8 encoded string
IntegerType	varint	Arbitrary-precision integer
Int32Type	int	4-byte integer
InetAddressType	inet	IP address string in xxx.xxx.xxx.xxx form
LongType	bigint	8-byte long
UUIDType	uuid	Type 1 or type 4 UUID
TimeUUIDType	timeuuid	Type 1 UUID only (CQL3)
DateType	timestamp	Date plus time, encoded as 8 bytes since epoch
BooleanType	boolean	true or false
FloatType	float	4-byte floating point
DoubleType	double	8-byte floating point
DecimalType	decimal	Variable-precision decimal
CounterColumnType	counter	Distributed counter value (8-byte long) (Only for column value)

Table 4.3.: Data Types in Cassandra

- ByteOrderedPartitioner : Keeps an ordered distribution of data lexically by key bytes.

The ByteOrderedPartitioner allows ordered scan by row key. For example if primary keys are username it is possible to scan rows for user whose name fall between Jack and John. However, using an ordered partitioner is not recommended for the following reasons : (i) Difficult load balancing (ii) Sequential writes can cause hot spots (iii) Uneven load balancing for multiple tables [Data].

The column names could be seen as SQL columns. In the Listing 4.2, column names are declared. However, it is not mandatory. If the column names cannot be known in advance, in case of dynamic tables, a default validation class can be specified instead of defining per column data types. An important aspect of column names is that they are *ordered* and are often also used as *values*.

Since Cassandra 0.8.1, it is possible to use composite column. Composite comparators can be considered as a comparator composed of several other types of comparators. Composite comparator provides the following benefits for data modeling : (i) custom inverted search indexes: allow more control over the column family than a secondary index (ii) a replacement for super columns (column of columns) (iii) grouping otherwise static skinny rows into wider rows for greater efficiency [Datb]. Figure 4.2 gives an example of a composite column with empty values for the column family "Timeline" that will be used in the third part of this thesis.

Finally, the values can be of all types listed in Table 4.3. When setting a column in Cassandra, it is possible to set an expiration time, or time-to-live (TTL) attribute.

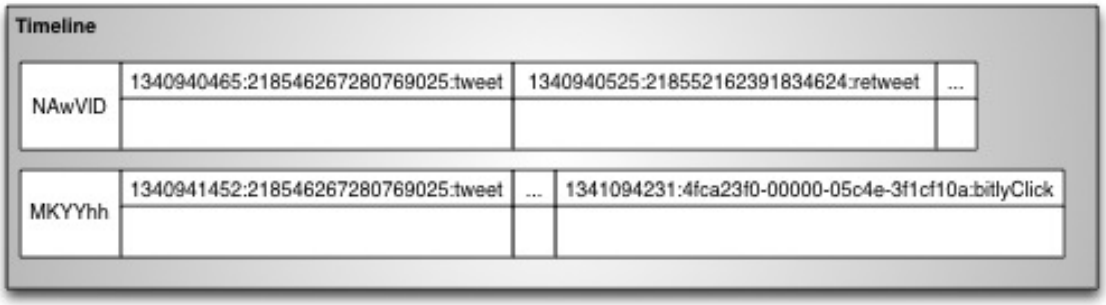


Figure 4.2.: Composite Column

4.4. Query Model

Queries on data can be done in different ways:

- Thrift API
  - CLI (Command Line Interface)
  - Higher level third-party librairies
    - Hector (java)
    - Pycassa (python)
    - Phpyandra (php)
    - Astynax (java)
    - Helenus (NodeJS)
- CQL (Cassandra Query Language)

Cassandra CLI is a very good tool for analyzing data and creating new keyspaces or column families on your cluster. The following command shows how to connect to Cassandra CLI :

```
$ cassandra-cli -host hostname -port 9160
```

The Listing 4.1 and Listing 4.2 show how to create a keyspace and a column family using CLI. When creating the keyspace, we specified the replica placement strategy (in this case "SimpleStrategy") and the replication factor.

When using a programming language to query or insert data into Cassandra, third-party libraries must be used. We will describe that which we will use in our use case : Hector, a java client for Cassandra.

The class in the Listing 4.3[1] shows how to insert a single column into Cassandra and make a query on this column.

[1]This example has been taken from : https://github.com/zznate/hector-examples/blob/master/src/main/java/com/riptano/cassandra/hector/example/InsertSingleColumn.java

```
public class InsertSingleColumn {

    private static StringSerializer stringSerializer = StringSerializer.get();

    public static void main(String[] args) throws Exception {
        Cluster cluster = HFactory.getOrCreateCluster("TestCluster", "localhost:9160");

        Keyspace keyspaceOperator = HFactory.createKeyspace("Keyspace1", cluster);
        try {
            Mutator<String> mutator = HFactory.createMutator(keyspaceOperator,
                StringSerializer.get());
            mutator.insert("jsmith", "Standard1", HFactory.createStringColumn("first", "John")
                );

            ColumnQuery<String, String, String> columnQuery = HFactory.createStringColumnQuery
                (keyspaceOperator);
            columnQuery.setColumnFamily("Standard1").setKey("jsmith").setName("first");
            QueryResult<HColumn<String, String>> result = columnQuery.execute();

            System.out.println("Read HColumn from cassandra: " + result.get());
            System.out.println("Verify on CLI with:  get Keyspace1.Standard1['jsmith'] ");

        } catch (HectorException e) {
            e.printStackTrace();
        }
        cluster.getConnectionManager().shutdown();
    }

}
```

Listing 4.3: Insert a single column using Hector

Finally, it is possible to use CQL to query Cassandra. CQL is an acronym for Cassandra Query Language and is really close to SQL. However as Cassandra is a NoSQL database it doesn't support joins or subqueries. The easiest way to make queries using CQL is to use cqlsh, a Python-based command-line client.

5 Hadoop

5.1. Introduction to Hadoop 26
5.2. Hadoop File System (HDFS) 27
5.3. MapReduce 28
5.4. Hadoop limitations 29

This chapter describes one of the most used tool for processing Big Data: Hadoop. Although Hadoop is really powerful and robust, it also suffers from some deficiencies.

5.1. Introduction to Hadoop

Hadoop is an open source framework for writing and running distributed applications which are capable of batch processing large sets of data. Hadoop framework is mainly known for MapReduce and its distributed file system (HDFS). The MapReduce algorithm, that consists of two basics operations : "map" and "reduce" is a distributed data processing model that runs on large cluster of machines.

Hadoop started out as a subproject of Nutch created by Doug Cutting. Nutch is itself an extension of Lucene, a full-featured text indexing and searching library. Nutch tried to build a complete web search engine using Lucene as its core component. Around 2004, Google published two papers describing the Google File System (GFS) and the MapReduce framework. Google claimed to use these technologies for scaling its own search systems. Doug Cutting immediately saw the applicability of these technologies to Nutch and started to implement the new framework and ported Nutch to it. The new version of Nutch boosted its scalability. Then Doug Cutting created a dedicated project that brought together an implementation of MapReduce and a distributed file system. Hadoop was born. In 2006, Cutting was hired at Yahoo! to work on improving Hadoop as an open source project [Lam11].

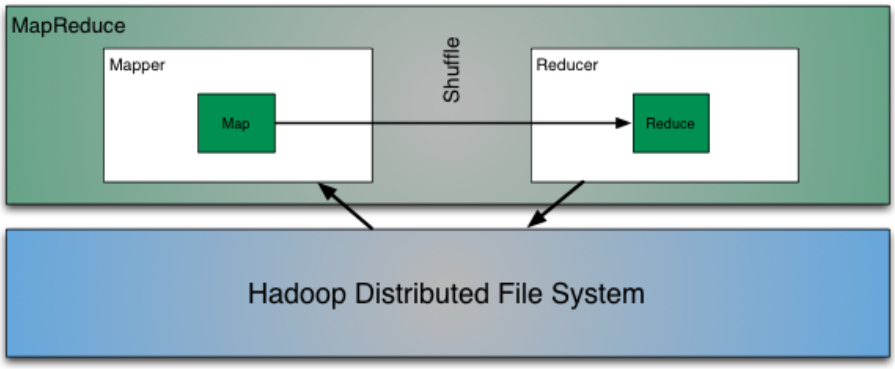


Figure 5.1.: Hadoop basic architecture

It is important to notice that the main goal of Hadoop is not to speed up data processing but to make possible to process really huge amount of data by splitting these data in smaller subsets of data. As shown in Figure 5.1, typically, Hadoop takes as input files in HDFS, does some computation with the MapReduce framework and outputs files in HDFS.

A cluster running Hadoop means running a set of daemons on the different servers of the network. The daemons include :

- NameNode : it is the master of HDFS that directs the slave DataNodes daemons. It is recommended to run the NameNode on it's own machine because the daemon is memory and I/O intensive. The negative aspect to the NameNode is that it is a single point of failure.
- DataNode : It is responsible of writing and reading HDFS blocs to actual files on the local filesystem. A DataNode may communicate with other DataNodes to replicate its data for redundancy.
- Secondary NameNode : it is an assistant daemon for monitoring purposes.
- JobTracker : Manages MapReduce job execution. It is the liaison between the application and Hadoop. The JobTracker daemon determines the execution plan, assigns nodes to different tasks. There is only one JobTracker per Hadoop cluster.
- TaskTracker : Manages the execution of individual tasks on the machine. Communicates to JobTracker in order to obtain task requests and provide task updates.

5.2. Hadoop File System (HDFS)

HDFS stands for Hadoop distributed file system. HDFS is a filesystem designed for large-scale distributed data processing under MapReduce. It is highly fault-tolerant and is designed to be deployed on low-cost hardware. The core abstraction in HDFS is the

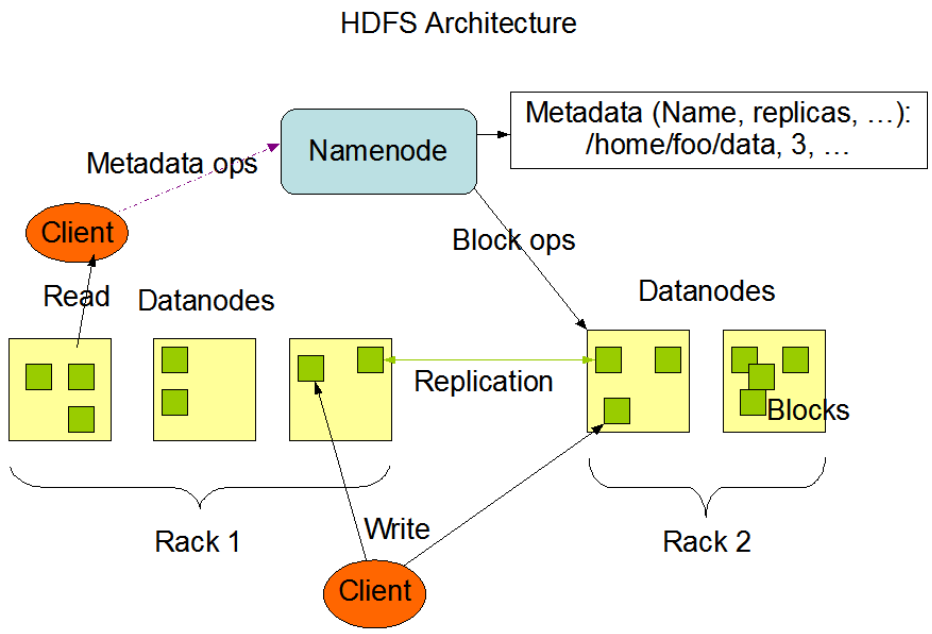


Figure 5.2.: HDFS architecture [Fouc]

file. Data is stored in blocks distributed and replicated over many nodes and it is capable to store a big data set, (e.g 100TB) as a single file. The block size is often range from 64MB to 1GB [Lam11].

Hadoop provides a set of command line utilities that work similarly to the basics Linux file commands. A command in Hadoop takes the form of : *hadoop fs -cmd <args>*. It is therefore possible to copy data from the local filesystem to HDFS. HDFS also provides an API with which data can be accessed and new data can be appended to existing files.

HDFS is written in Java and was designed for mostly immutable files and may not be suitable for systems requiring concurrent write operations.

The Figure 5.2 gives an overview of the most important HDFS components.

5.3. MapReduce

MapReduce is a data processing model. Its biggest advantage is the easy scaling of data processing over many computing nodes. The primitives to achieve data processing with MapReduce are called mappers and reducers. The data model is quite strict and, sometimes, decomposing a data processing application into mappers and reducers is not so easy. But once the application is written in the MapReduce form, scaling it to run over thousands of machines in a cluster is trivial [Lam11].

The Figure 5.3 shows a very common example of a MapReduce job: word count. A MapReduce job takes as input files in HDFS. The mapper is responsible for splitting

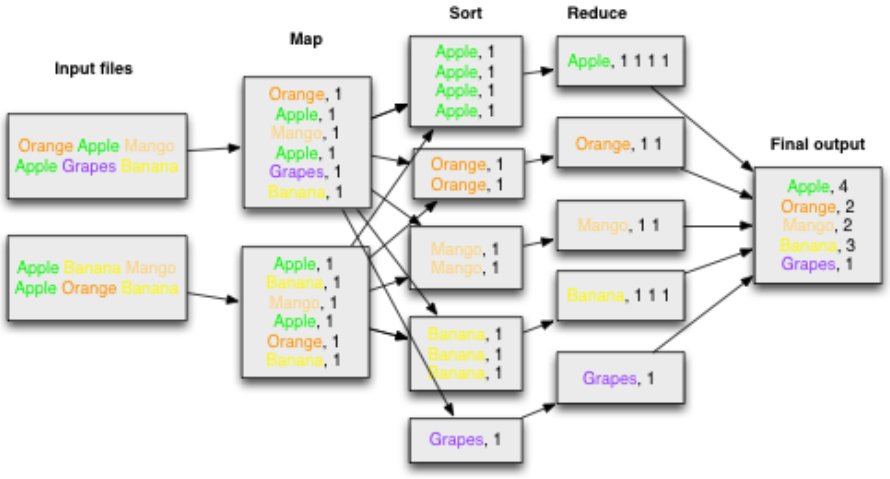


Figure 5.3.: MapReduce word count example

lines in the files word by word. Then the data model in MapReduce is based on key - value pairs. The word (the key) is mapped with a value (the number one in this example). The last step in the mapper is done by the partitioner. The partitioner decides the target reducer for the key value pairs. Then, in the mapper the key value pairs are sorted and grouped by key. Finally, the final result is composed of files in HDFS.

5.4. Hadoop limitations

Actually, Hadoop does its job very well; processing huge set of data. But the model has some limitations. Firstly, the data model is quite restrictive, it is not always easy to transform our problems into key value pairs. Moreover, Hadoop can only process a finished set of data, it means that MapReduce can only process data by batch. Therefore, when a batch is finished, data is already aged by, at least, the time required by the batch. Hadoop is definitely not a good fit for processing the latest version of the data.

The Figure 5.4 shows this limitation of processing data by batch. Once the "Batch1" is terminated data available for the end-user is aged by a least "t1", until the second batch will be terminated.

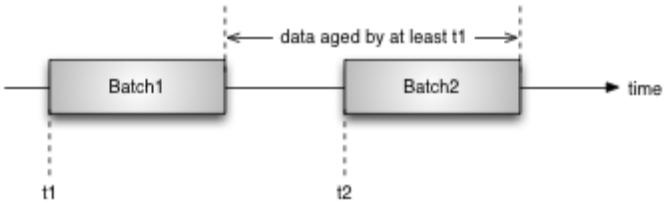


Figure 5.4.: MapReduce is limited to batch processing

Part II.

Real time data processing and Storm

6

Real time data processing and Big Data

6.1. Real time data processing 32

6.2. Real-time data processing systems 34

6.2.1. MapReduce Online 35

6.2.2. Apache Drill 35

6.2.3. Cloudera Impala 37

6.2.4. StreamBase 37

6.2.5. IBM InfoSphere 37

6.2.6. Truviso 38

6.2.7. Yahoo S4 38

6.2.8. Storm 38

6.3. The Lambda architecture 39

This chapter describes the challenges of real time data processing in the context of Big Data.

6.1. Real time data processing

As seen before, Hadoop is the most widely used tool for processing large data sets. However, Hadoop is limited to processing data by batch. When a job is terminated, data produced are already aged by, at least, the time required by the batch. A solution would be to add always more nodes to reduce the batch time or get the batch size ever smaller and smaller. But when you get to the batch size of one it doesn't make any sense. Anyway Hadoop will never be a good fit for real-time computing as Yahoo CTO Raymie

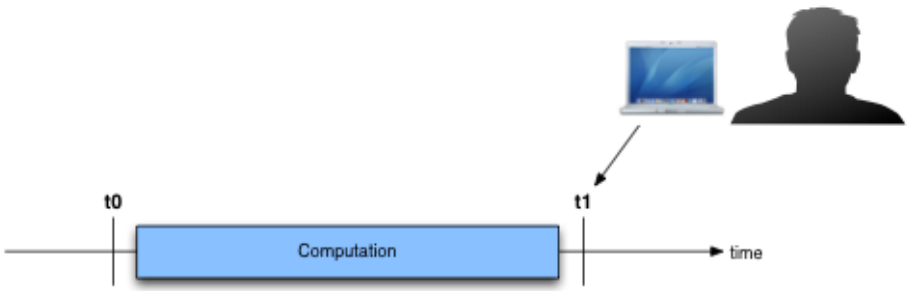


Figure 6.1.: Real-time : processing the latest version of the data

Stata explained : "With the paths that go through Hadoop [at Yahoo!], the latency is about fifteen minutes. [...] It will never be true real-time. It will never be what we call "next click," where I click and by the time the page loads, the semantic implication of my decision is reflected in the page" [Met].

So what can be considered as "real-time"? Firstly, it is important to emphasize that this term can have two different meanings. Real-time can be considered from the point of view of the data or from the point of view of the end-user.

Real-time from the point of view of the data means processing the latest version of the data, processing the data as it arrives. The Figure 6.1 shows this notion of real-time, processing data as it arrives. At time t_0, an event happens, at time t_1 the event is completely processed by the system and readable for the end-user or the application. What is the maximum time between t_0 and t_1 for which we can say that the event is processed in real-time? Actually the answer differs for each situation and there is no perfect definition of what should be true real-time. For instance, if a real-time system is used when the driver pushes the brake, analyze the situation and brakes. It will be better that, in this situation, it means a few milliseconds. In contrast, if a real-time system is used to detect trends in Twitter stream, the notion of real-time can be delayed by a few seconds. Nowadays, this term is quite a buzzword and is often misused. But generally, when speaking about real-time system, the maximum latency shouldn't exceed a few seconds.

The second notion of real-time, from the point of view of the end-user means getting a response of a query in real-time (i.e with low latency) as shown in Figure 6.2. The difference between t_1 and t_0 should be as small as possible. This notion of real-time can be compared as REST services or any other RPC, as opposed to batch processing when the output is available only some minutes or hours later [Per13b].

Before exploring solutions for doing real-time computing let's analyze why having informations in real-time could be a great opportunity. Processing data as it arrives give the opportunity to have information about what happen now in the business. For example, it

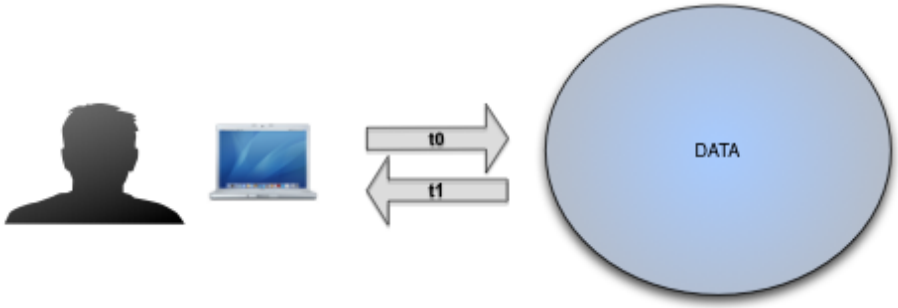


Figure 6.2.: Real-time : getting response of query in real-time

is possible to aggregate data and extract trends about the actual situation. It is possible to know how many products have been sold during the last few minutes, analyze stocks and take decisions in real-time, integrate data in a centralized data warehouse, etc.

Being able to process data with low latency, almost immediately is also really useful for different possibilities : for example it is possible to provide recommendations for an user on a website based on its history. Huge amount of data can be processed and the results are available for the end user almost in real-time.

More generally, some questions are essential for a company. The first is *what happens in the business*, bring to the light the business situation, trends, patterns and exceptions. Then it is as important to know *why it happens* and finally the last question would be *what is likely to happen in the future* [MS]. Being able to answer these questions is already good and not easy but answering these questions based on what is happening now will definitely be the panacea. Taking the right decision at the right moment could be a real advantage. A lot of answers are present in the data, the goal is to ask the right questions and have the right tools to meet them.

Even if there is no standard system for real-time computing yet, a lot of systems with different approaches are emerging. The following section will give an overview of these systems.

6.2. Real-time data processing systems

Firstly, it is important to distinguish what is the goal of a real-time system in the context of Big Data: process data in real-time or get response in real-time. Processing data in real-time is generally called "stream processing". A stream is an unbounded source of events. The goal of a stream processing system is to do transformation on these streams of data in real-time and to make processed data directly available for the end-user or the application.

On the oder hand, nowadays, a lot of systems try to process large amount of data with low latency, almost in real-time, as opposed to Hadoop and MapReduce for example.

This chapter will give an overview of these tools from both side: stream processing and low latency response.

From the side of stream processing, two main systems are emerging : Storm and Yahoo S4. Storm tends to become the standard tool for stream processing. Apache Drill and Cloudera Impala are both based on Google Dremel and try to process petabyte of data and trillions of records in seconds. Other approaches, like MapReduce Online, tries to improve MapReduce to get early results of the batch.

6.2.1. MapReduce Online

MapReduce Online proposes a modified MapReduce architecture that allows data to be pipelined between operators. This extends the MapReduce programming model beyond batch processing and supports online aggregation, which allows users to see early returns from a job as it is being computed. The Hadoop Online Prototype (HOP) also supports continuous queries, therefore applications such as event monitoring and stream processing can be written [TC09].

The biggest difference between the original MapReduce algorithm and MapReduce Online is that intermediate data is no more temporary materialized in a local file (on each node) but is pipelined between operators, while preserving the programming interfaces and fault tolerance models of previous MapReduce frameworks.

To validate this architecture, a pipelined version of Hadoop was designed, the Hadoop Online Prototype. Pipelining provides many advantages [TC09]:

- Since reducers begin processing data as soon as it is produced by mappers, they can generate and refine an approximation of their final answer during the course of execution.
- Pipelining extends the domain of problems to which MapReduce can be applied, especially for event monitoring and stream processing.
- Pipelining boosts the performance. Data is delivered to downstream operators more promptly, which can increase the possibilities for parallelism, improve utilization, and reduce response time.

6.2.2. Apache Drill

"Apache Drill (incubating) is a distributed system for interactive analysis of large-scale datasets, based on Google's Dremel. Its goal is to efficiently process nested data. It is a

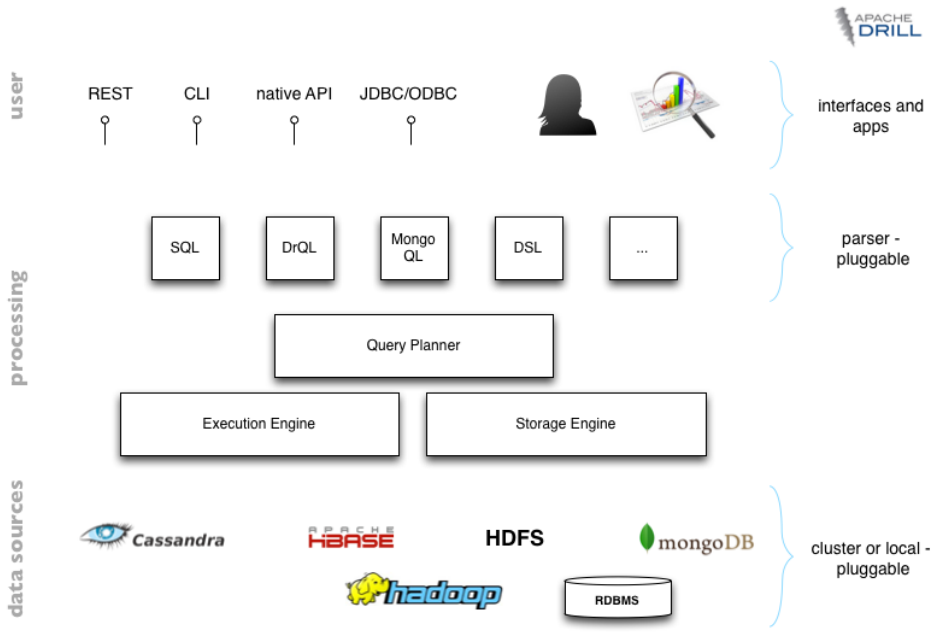


Figure 6.3.: Overview of Drill's architecture

design goal to scale to 10,000 servers or more and to be able to process petabyes of data and trillions of records in seconds" [Foua].

It is important to not be confused, Apache Drill is not a tool to process a stream of data, process data in real-time. But its goal is to get a response of queries on big data sets, almost in real-time.

Figure 6.3 gives an overview of the high-level architecture of Drill. There are essentially three parts [Foua]:

- User part : Allows direct interaction with interfaces such as command line interface (CLI) and a REST interface. Or allows application to interact with Drill.
- Processing part : Drill's core and pluggable query languages (needs parser that can turn the respective query language into Drill's logical plan).
- Data source part : pluggable data sources, either local or in a cluster setup.

The basic execution flow of Drill is the following : A user or machine generated query is created and handed to a query parser. Then the query is converted into a logical plan which describes the basic dataflow of the query operation. The logical plan is converted into a physical (or execution) plan through an optimizer and the execution engine is responsible for executing the plan.

6.2.3. Cloudera Impala

Impala is another tool based on Google Dremel and is therefore quite similar to Apache Drill. Its main goal is to query data, whether stored in HDFS or HBase in real-time. Impala uses the same metadata, SQL syntax, ODBC driver and user interface as Apache Hive, providing a unified platform for both batch oriented and real-time queries [MK].

In order to avoid latency, Impala circumvents MapReduce to directly access the data through a specialized distributed query engine that is quite similar to those found in parallel RDBMSs [MK].

6.2.4. StreamBase

StreamBase is a company founded in 2003 to commercialize a project called "Aurora" developed by Mike Stonebraker at MIT. StreamBase has been recently acquired by TIBCO software on June 11, 2013 and the company name will now be TIBCO StreamBase.

StreamBase is mainly composed of two commercial products : StreamBase LiveView and StreamBase CEP. StreamBase LiveView is a "push-based real-time analytics solution that enables business users to analyze, anticipate and alert on important business events in real-time, and act on opportunities or threats as they occur [Str]." The main use cases for StreamBase LiveView are mainly in financial domains : trade execution consulting, trade risk management, FX liquidity analysis, transaction cost analysis, etc [Str].

StreamBase CEP (complex event processing) combines a visual application development environment and an low-latency high-throughput event server. StreamBase CEP can be used to develop application for : algorithmic trading, pricing and analytics, fraud detection, market data management, ... [Str]

6.2.5. IBM InfoSphere

IBM InfoSphere is a commercial product that enables a set of tools for information integration and management [Wik13c].

The main tools include:

- IBM InfoSphere Information Server: is the core data integration
- IBM InfoSphere Streams: "is an advanced computing platform that allows user-developed applications to quickly ingest, analyze and correlate information as it arrives from thousands of real-time sources. The solution can handle very high data throughput rates, up to millions of events or messages per second" [IBM].
- IBM InfoSphere BigInsights: is a tool on the top of Hadoop. It simplifies the use of Hadoop through "Big SQL", a SQL interface for Hadoop

- IBM InfoSphere Warehouse: is a data warehousing and analytics software platform based on IBM DB2, a scalable relational database

IBM provides a fully integrated solution that contains a classical data warehouse, an Hadoop platform and a stream processing engine.

6.2.6. Truviso

Even if Truviso seems to be a quite promising tool there is not a lot of information about it. "Truviso is a continuous analytics, venture-backed, startup headquartered in Foster City, California developing and supporting its solution leveraging PostgreSQL, to deliver a proprietary analytics solutions for net-centric customers." [Wik13d]

Truviso combines a stream processing engine, a Java based integration framework and a visualization front end based on Adobe Flex. The processing engine is embedded into the open-source PostgreSQL database, uses SQL to continuously query events as they flow past[Din07].

In order to enable SQL to query infinite data streams, Truviso extended it in two ways. First it adds windowing capabilities to restrict the amount of data that it would be exposed to in any instant. Then, Truviso adds some syntactic to define event patterns that we might want to expect[Din07].

Truviso was acquired by Cisco on May 4, 2012.

6.2.7. Yahoo S4

As Storm, Yahoo S4 is a tool for processing streams in real-time. S4 was initially released by Yahoo! in October 2010. It is now an Apache Incubator project since September 2011 and is therefore licensed under the Apache 2.0 license.

S4 is a general purpose, near real-time, distributed, decentralized, scalable, event-driven, modular platform that allows programmers to easily implement applications for processing continuous unbounded streams of data [Foud].

6.2.8. Storm

Storm is probably the open-sourced tool the most used for streams processing. Storm will be described more deeply in chapter 7.

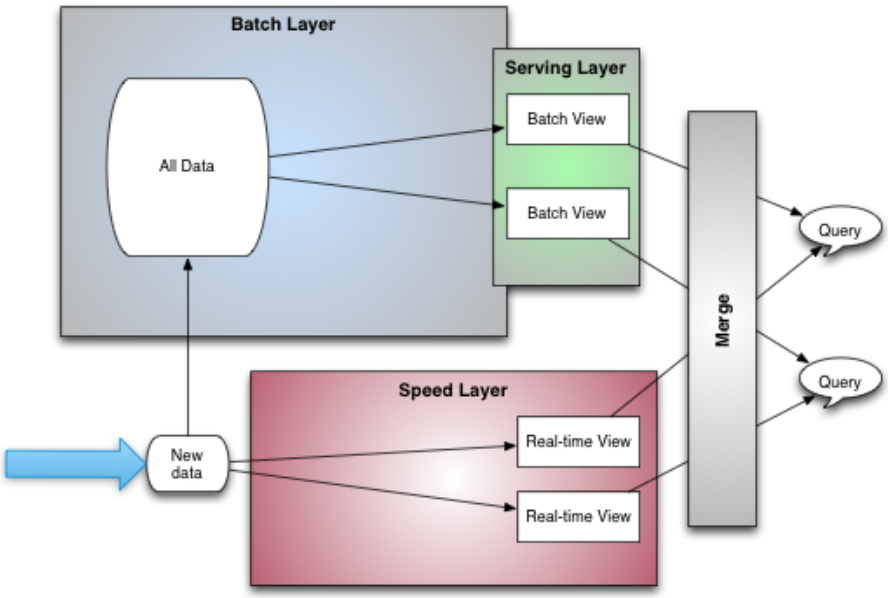


Figure 6.4.: The lambda architecture

6.3. The Lambda architecture

We have seen how Hadoop works and its batch processing approach. We have also had an overview of stream processing tools. Now let's analyze an approach that mixes both approaches, batch processing and real-time data processing. This approach is called the "Lambda Architecture" and has been created by Nathan Marz, the main developer of Storm.

This approach is divided in three layers :

- The batch layer
- The serving layer
- The speed layer

The Figure 6.4 gives an overview of this approach.

All new data is sent to both the batch and the speed layer. In the batch layer, new data is appended to the master data set. The master dataset is a set of files in HDFS and contains the rawest information that is not derived from any other information. It is an immutable append-only set of data [Mar12].

The batch layer precomputes query functions from scratch continuously, in a "while(true)" loop. The results of the batch layer are called "batch views" [Mar12].

The serving layer indexes the batch views produced by the batch layer. Basically, the serving layer is a scalable database that swaps in new batch views as they are made

available. Due to the latency of the batch layer, the results available from the serving layer are always out of date by a few hours [Mar12].

The speed layer compensates for the high latency of updates to the serving layer. This layer uses Storm to process data that have not been taking into account in the last batch of the batch layer. This layer produces the real-time views that are always up to date and stores them in databases that are good for both read and write. The speed layer is more complex than the batch layer but that complexity is compensated by the fact that the realtime views can be continuously discarded as data makes its way through the batch and serving layers [Mar12].

Finally, queries are resolved by merging the batch and real-time views.

This approach becomes quite popular and has many benefits. First, the whole approach can be built with open-source tools, that means no license costs. Then the batch views are always recomputed from scratch, it is therefore possible to adjust the granularity of the data in function of its age. An other benefit of recomputing data from scratch is that if the batch views or the real-time views are corrupted, as the main data set is append only, it is easy to restart from scratch and recover from the buggy state. Last but not least, the end-user can always query the latest version of the data due to the speed layer [Per13b].

7

Introduction to Storm

7.1. What is Storm? 41
7.2. Components of a Storm cluster 42
7.3. Streams and Topologies 42
7.4. Parallelism hint 44
7.5. Stream groupings 45
7.6. Fault-tolerance in Storm 45
7.7. Guaranteeing message processing 47
7.7.1. Transactional topologies 50
7.8. Trident 50
7.9. Storm UI 51

This chapter describes the basis of Storm. This chapter is mainly inspired from the Storm's wiki available at the address https://github.com/nathanmarz/storm/wiki

7.1. What is Storm?

Storm is a *distributed real-time computation system* that is fault-tolerant and guarantees data processing. Each of these key words will be describe later in this chapter. Storm was created at Backtype, a company acquired by Twitter in 2011. It is a free and open source project licensed under the Eclipse Public License. The EPL is a very permissive license, allowing you to use Storm for either open source or proprietary purposes. Storm makes it easy to reliably process unbounded streams of data, doing for realtime processing what Hadoop did for batch processing. Storm is very simple and was designed from the ground up to be usable with any programming language.[Marb]

Storm can be used for a lots of different use cases :

1. **Stream processing** : Storm can be used to process a stream of new data and update databases in realtime.
2. **Continuous computation** : Storm can do a continuous query and stream the results to clients in realtime.
3. **Distributed RPC** : Storm can be used to parallelize an intense query on the fly.

If setup correctly storm can also be very fast: a benchmark clocked it at over a million tuples processed per second per node [Marc].

7.2. Components of a Storm cluster

A Storm cluster can be compared to a Hadoop cluster. Whereas on Hadoop you run "MapReduce jobs", on Storm you run "topologies". The main difference between "jobs" and "topologies" is that a MapReduce job eventually finishes and processes a finished dataset, whereas a topology processes messages forever (or until you kill it) as they arrive, in real-time [Mar11].

There are two kinds of nodes on a Storm cluster, the master and the worker nodes. The master node runs a daemon called "Nimbus". Nimbus can be compared to the Hadoop's JobTracker and is responsible for distributing code around the cluster, assigning tasks to machines and monitoring failures.

Each worker node runs a daemon called the "Supervisor". The supervisor listens for work assigned to its machine and manages worker processes as necessary, based on what Nimbus has assigned to it. Each worker process executes a subset of a topology. A running topology consists of many worker processes distributed across many machines.

All coordination between Nimbus and the Supervisors is done through Zookeeper. Zookeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization and group services.

7.3. Streams and Topologies

The core abstraction in Storm is the "stream". A stream is just an unbounded sequence of tuples. A **tuple** is a named list of values, and a field in a tuple can be an object of any type. Storm provides the primitives for transforming a stream into a new stream in a distributed and reliable way.

The role of a storm project is to process streams. In order to process these streams we have spouts and bolts.

A **spout** is simply a source of streams. A spout can read tuples from different sources, for example a queue system (like Kestrel or Kafka) an API or simply a plain text file,

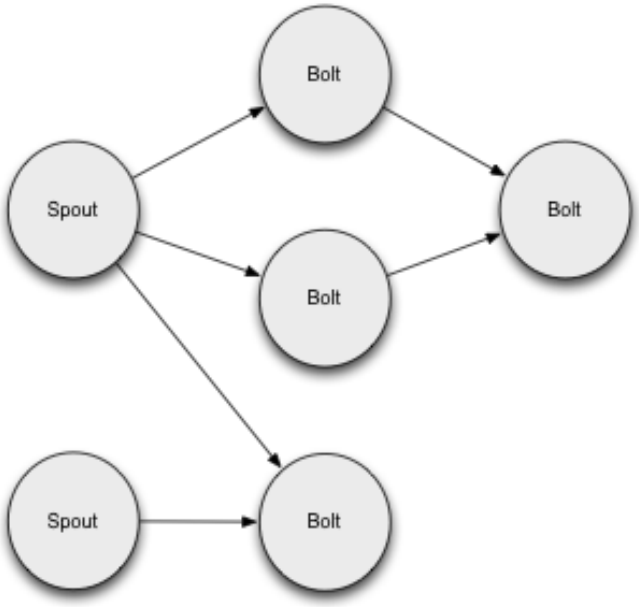


Figure 7.1.: An example of a simple Storm Topology

and emit them as a stream. A lot of different spouts are available at the address https://github.com/nathanmarz/storm-contrib

A **bolt** does single-step stream transformations. A bolt consumes any number of streams, does some computation and possibly emits new streams. A bolt can receive its stream from a spout or another bolt.

The spouts and bolts are packaged into a "topology" which is the top-level abstraction that you submit to Storm clusters for execution. A topology is a graph of stream transformations where each node is either a spout or a bolt. A topology representation is shown in Figure 7.1. Edges in the graph indicate which bolts are subscribing to which streams.

In Storm, and in stream processing in general it is really important that the tasks are able to process the data at a rate higher than new data arrives. Otherwise, the tuples will wait to be processed and possibly timeout and failed to be processed. Hopefully, each process in a Storm topology can be executed in parallel. In a topology, we can specify for each process how much parallelism we want, and then Storm will spawn that number of thread across the cluster in order to do the execution.

The notion of parallelism is not so easy to understand. Therefore, it will be described deeper in the next section.

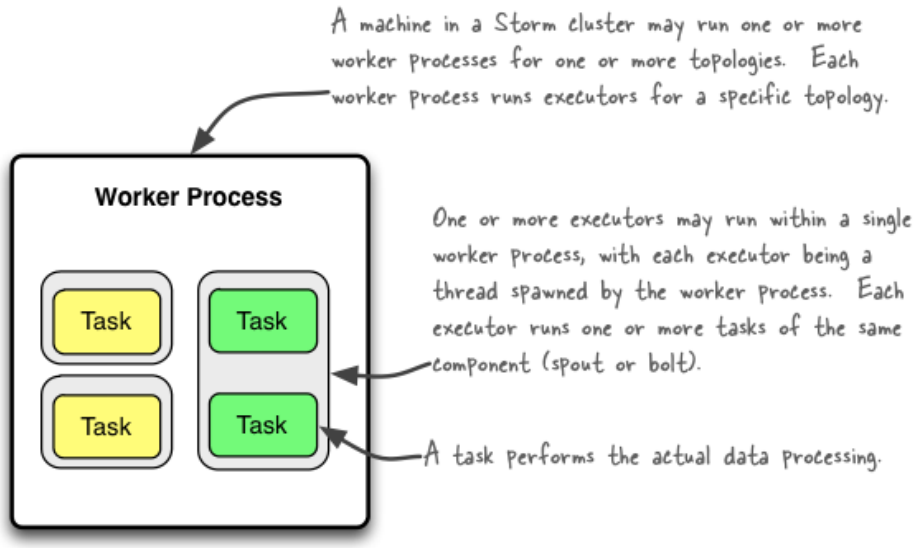


Figure 7.2.: Relationships of worker processes, executors and tasks. [Nol12]

7.4. Parallelism hint

The following example shows how the parallelism hint is specified for a bolt :

```
builder.setBolt("BoltName", new BoltName(), 2)
  .shuflleGrouping("PreviousBolt");
```

First with the method setBolt we add a new bolt to the topology, we specify a name for this bolt and a new instance of this bolt is declared. The number "two" represents the number of parallelism that we want to use for this bolt. The notion of grouping will be covered later in this chapter.

In order to run a topology, Storm distinguishes three main entities :

- Worker processes
- Executors
- Tasks

The Figure 7.2 describes the relationships between these entities.

A worker process belongs to a specific topology and executes a subset of a topology. Many worker processes may be run on a machine in a Storm cluster. Each worker process runs executors for a specific topology [Nol12].

An executor is nothing else than a thread spawned by a worker process. It may run one or more tasks for the same spout or bolt. The number of executors for a component can change over time. What we called parallelism hint in Storm actually corresponds to the number of executors, so the number of threads.

Finally, the tasks perform the data processing. Even if the number of executors for a component can change over time, the number of tasks for a component is always the same as long as the topology is running. By default, Storm will run one task per thread [Nol12].

In the beginning of this section we saw how to specify the number of parallelism hints, so the number of executors (threads) for a specific bolt. In order to specify the number of worker processes to create for the topology across machines, the method "setNumWorkers" of the class config must be used.

For specifying the number of tasks to create per component, we should use the method "setNumTasks" as shown in the following sample code :

```
builder.setBolt("BoltName", new BoltName(), 2)
  .setNumTasks(4)
  .shuflleGrouping("PreviousBolt");
```

In the above example, Storm will run two tasks (4/2) per executor, i.e. per thread.

The understanding of parallelism in a Storm topology should now be clear. But another question arises: how to send tuples between set of tasks? To which task of a Bolt a specific tuple should go? The next section will answer this question.

7.5. Stream groupings

To introduce the notion of stream grouping, let's take an example. We could imagine that a bolt sends tuples with the following fields : timestamp, webpage. This could be a simple clickstream. The next bolt which has a parallelism hint of four is responsible for counting the clicks by webpage. It is therefore mandatory that the same webpages go to the same task. Otherwise, we will need to merge the results in a further process and this is not the goal. Therefore, we need to group the stream by the field webpage.

The different kinds of stream grouping are described in the Table 7.1.

In most cases shuffle and fields grouping will be mainly used.

It is also possible to implement our own stream grouping by implementing the "CustomStreamGrouping" interface.

7.6. Fault-tolerance in Storm

Fault-tolerance is the property that enables a system to continue operating properly in the event of failure of some of its components [Wik13b]. Especially in case of distributed system, where big clusters can be composed of hundreds or even thousand nodes, it is essential that the system can handle failures. The list of things that can go wrong is

Name	Description
Shuffle grouping	Tuples are randomly distributed across the bolt's tasks in a way such that each bolt is guaranteed to get an equal number of tuples.
Fields grouping	The stream is partitioned by the fields specified in the grouping.
All grouping	The stream is replicated across all the bolt's tasks. Use this grouping with care.
Global grouping	The entire stream goes to a single one of the bolt's tasks. Specifically, it goes to the task with the lowest id.
None grouping	This grouping specifies that you don't care 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).
Direct grouping	This is a special kind of grouping. A stream grouped this way means that the producer of the 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 the 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 the task ids that the tuple was sent to).
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.

Table 7.1.: Built-in Stream grouping [Marb]

almost unbounded. It can be hardware issues, for example a hard disk can be broken or the fan doesn't work anymore etc. Problems can also come from the network, the software, user's code, etc.

How does Storm react in case of failures? In order to answer this question, let's analyze the different situations that could happen.

First, if a worker dies, the supervisor will restart it. But if it continuously fails and is unable to heartbeat to Nimbus, Nimbus will reassign this worker to another machine [Marb].

If the whole node dies, the tasks assigned to that machine will time-out and Nimbus will reassign those tasks to other machines [Marb].

If Nimbus or Supervisor deamons die, no worker processes will be affected. This is in contrast to Hadoop, where if JobTracker dies, all the running jobs are lost. To ensure that the system works properly, Nimbus and Supervisors deamons must be run under supervision. So in case they die, they simply restart as if nothing happened. Storm is a fail-fast system, which means that the processes will halt whenever an unexpected error is encountered. Storm is designed so that it can safely halt at any point and recover correctly when the process is restarted. This is why Storm keeps no state in-process. Therefore, if Nimbus or the Supervisors restart, the topologies are unaffected [Marb].

In our use cases, the tool used to supervise processes is called "supervisord". Supervisord is a client/server system that allows its users to control a number of processes. Don't be confuse, even if the name is the same, "Supervisor" the tool used to supervise processes has nothing to do with the Storm's deamon called supervisor!

Therefore, Nimbus is not really a single point of failure. If the Nimbus node is down, the workers will still continue to function. Additionally, supervisors will continue to restart workers if they die. However, without Nimbus, workers won't be reassigned to other machines when necessary [Marb].

7.7. Guaranteeing message processing

In a topology, it is possible to guarantee that each message coming off a spout will be fully processed. First, let's define the term "fully processed". The spout tuple (the tuple coming off the spout) can spawn many tuples. Consider the following example :

```
1 TopologyBuilder builder = new TopologyBuilder();

3 builder.setSpout("lines", new readFileSpout(), 1);
builder.setBolt("split", new splitBolt(), 2).
5   shuffleGrouping("lines");
builder.setBolt("count", new countBolt(), 4).
7   fieldsGrouping("split", new Fields("word"));
```

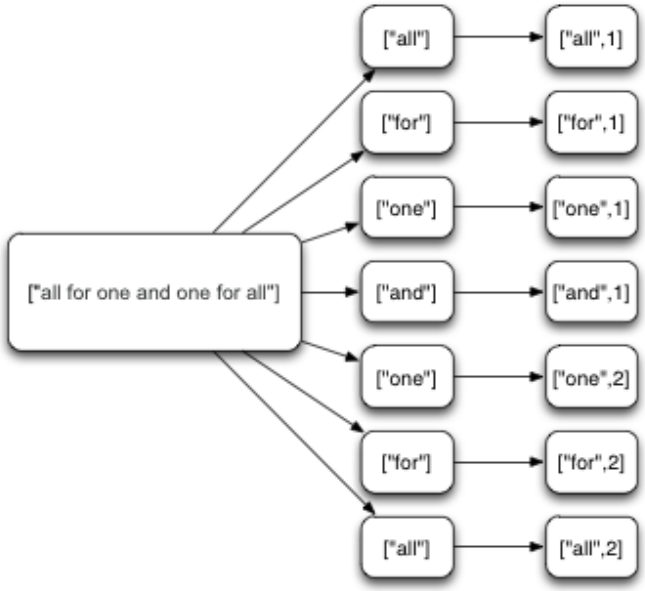


Figure 7.3.: Tree of tuples

This simple sample topology reads a file line by line. Each line is emitted as a tuple. A bolt splits the lines word by word and the last bolt emits, for each word, the number of times it has been seen. The tuple tree will look like the Figure 7.3 [Marb].

Storm considers a spout tuple fully processed when every tuple in the tree has been processed. A spout tuple is considered as failed when its tree of tuples fails to be processed within a specified timeout. This timeout can be configured (carefully) and is 30 seconds by default. In other words, if the spout tuple is still in the topology after the timeout, it will fail [Marb].

The following listing represents the interface that spouts implement :

```
1 public interface ISpout extends Serializable {
    void open(Map conf, TopologyContext context, SpoutOutputCollector collector);
3   void nextTuple();
    void close();
5   void ack(Object msgId);
    void fail(Object msgId);
7 }
```

First, Storm processes tuple after tuple in the nextTuple() method. The spout uses SpoutOutputCollector provided in the open method to emit a tuple and for each tuple a message id. Emitting a tuple in the nextTuple method looks like :

```
1 _collector.emit(new Values("fields1", "fields2", 3), msgId);
```

Then the tuple is consumed by the bolts and Storm tracks the tree of tuples generated. If Storm detects that a tuple is fully processed, it will call the ack method on the originated spout task with the message id that the spout provided. In contrast, if the tuple times-out, Storm will call the fail method on the spout [Marb].

For example when using a queue messaging system, tuples in the topology will be in a pending state in the queue. If the tuple is fully processed the ack method removes the tuple from the queue. Otherwise the fail method tells the queue that the message with a specific id needs to be replayed.

The user needs to do two things in order to benefit from Storm reliability capabilities. First, tell Storm whenever a new link in the tree of tuples is created. Second, tell Storm when a tuple's process is over. Therefore, Storm can detect when the tree of tuples is fully processed and can ack or fail the spout in consequence [Marb].

Specifying a link in the tuple tree is called *anchoring*. Anchoring is done when the tuple is emitted. The following listing represents the "execute" method of a bolt in which tuples are anchored when they are emitted:

```
public void execute(Tuple tuple){
  String sentence = tuple.getString(0);

  for(String word: sentence.split(" ")) {
    _collector.emit(tuple, new Values(word));  //Specifying the input tuple as the first
        argument to emit anchored tuple
  }
  _collector.ack(tuple); //Ack the tuple in the tree of tuple
}
```

Since the word tuple is anchored, the spout tuple at the root of the tree will be replayed later on if the word tuple failed to be processed downstream. In contrast, the following listing emit an unanchored tuple :

```
_collector.emit(new Values(word));
```

Therefore, if the tuple fails to be processed downstream, the spout tuple (the root tuple) won't be replayed. Depending on the use case, it can be useful to emit unanchored tuples.

Finally, we need to specify when a tuple's process is over. This is done by using the *ack* and *fail* method on the OutputCollector. The fail method can be called to immediately fail the spout tuple. For example, if the tuple failed to be stored in the database we can explicitly fail the input tuple. Therefore, the tuple can be replayed faster than waiting for the tuple to time-out.

It is essential that each tuple is either acked or failed. Storm uses memory to track the tuples. Therefore, if tuples are not acked or failed, the task will eventually run out of memory.

Storm has an interface called BasicBolt that automatically anchors the tuple when emitting and acks it at the end of the "execute" method. The split sentence example can be written as a BaseBasic bolt like the following listing :

```
public class SplitSentence extends BaseBasicBolt {
  public void execute(Tuple tuple, BasicOutputCollector collector) {
    String sentence = tuple.getString(0);

    for(String word: sentence.split(" ")){
```

```
    collector.emit(new Values(word));
    }
  }
  public void declareOutputFields(OutputFieldsDeclarer declarer){
    declarer.declare(new Fields("word"));
  }
}
```

The execute method does exactly the same as the previous example, the tuple is anchored when emitted and acked at the end of the execute method.

In order to track the tree of tuples, Storm has a set of special "acker" tasks. When an acker sees that the tree is fully processed it sends a message to the spout task that created the spout tuple. By default, Storm uses one acker task but this number can be specified in the topology configuration using config.TOPOLOGY_ACKERS. It is generally recommended to setup the number of ackers equal to the number of workers [Marb].

7.7.1. Transactional topologies

Storm guarantees that the tree of tuple for a specific spout tuple will be fully processed. Now let's say that a part of the tuple of the tree has been processed, and then a tuple of the tree fails to be processed. Therefore, the spout tuple will be replay and the tuples that have already been processed will be processed again. It can be a problem if, for example we want to count the occurrences of words in a sentence. Some words will be counted more than once.

A transactional topology guarantees that tuples will be processed exactly once.

7.8. Trident

Trident is a high-level abstraction on top of Storm like Pig or Hive in Hadoop. Trident contains joins, aggregations, grouping, functions and filters. In addition Trident adds primitives for doing stateful incremental processing on top of any database or persistent store [Marb].

Let's look at an illustrative example of Trident that will do two things [Marb]:

1. Compute streaming word count from an input stream of sentences
2. Implement queries to get the sum of the counts for a list of words

The following listing represents the implementation of a such wordcount example :

```
TridentTopology topology = new TridentTopology();
TridentState wordCounts =
     topology.newStream("spout1", new spout())
       .each(new Fields("sentence"), new Split(), new Fields("word"))
```

```
        .groupBy(new Fields("word"))
        .persistentAggregate(new MemoryMapState.Factory(), new Count(), new Fields("count"))
        .parallelismHint(6);
```

First, a TridentTopology object is created. Then, TridentTopology has a method called "newStream" that simply creates a new stream of data into the topology reading from an input source. An input source can be from different types as in a typical Storm topology. For example it can be queue brokers like Kestrel of Kafka or a simple spout that reads information from an external source [Marb].

Then Trident processes the stream as small batches of tuples. Depending on the incoming throughput, the size of batches can vary from thousands to millions of tuples [Marb].

Back to our example, the spout emits a stream that contains one field called "sentence". The next line applies the split function that is responsible for splitting the sentence word by word. The following listing describes the implementation of the split function :

```
public class Split extends BaseFunction {
   public void execute(TridentTuple tuple, TridentCollector collector) {
       String sentence = tuple.getString(0);
       for(String word: sentence.split(" ")) {
           collector.emit(new Values(word));
       }
   }
}
```

The implementation is really straightforward. The class extends "BaseFunction" and the execute method simply splits the string sentence and emits each word.

The stream is grouped by the field called "word" and then each group is persistently aggregated using the Count aggregator. The persistentAggregate function knows how to store and update the results of the aggregation in a source of state. In this example, the word counts are kept in memory but it would have been really easy to use Cassandra, for example. The values stored by persistentAggregate represent the aggregation of all batches ever emitted by the stream [Marb].

Finally, the last line specifies the number of parallelism hints to use.

7.9. Storm UI

Storm UI is a web interface that shows information about Storm's cluster and running topologies. It is the good place to analyze the performance of our running topologies and see what's going wrong in case of problems.

In order to enable Storm UI, it is necessary to run a deamon called "UI" on the nimbus host. The Storm UI is accessible to the url : http://{nimbus host}:8080

Part III.

Storm : use cases

8

Input data

8.1. Twitter **53**

8.1.1. Public streams 54

8.1.2. Data format 54

8.1.3. Rate limit 55

8.1.4. Other providers of Twitter data 55

8.2. Bitly **55**

8.3. Link between Twitter and Bitly **57**

This chapter describes the type of data that is mainly used for running Storm topologies.

8.1. Twitter

Twitter provides a full range of tools for developers. As Storm is mainly used as a stream processing system, we first tried to use, the streaming API. The sets of streaming APIs offered by Twitter give developers low latency access to Twitter's global stream of Tweet data.

Twitter offers several streaming endpoints [Twi]:

- Public streams : streams of the public data flowing through Twitter. Suitable for following specific topics, users and data mining.
- User streams : containing roughly all the data corresponding with a single user's view of Twitter.
- Site streams : multi-version of user streams. Site streams are intended for servers which must connect to Twitter on behalf of many users.

Parameter	Description
follow	A comma separated list of user IDs, indicating the users to return statuses for in the stream.
track	Keywords to track. Phrases of keywords are specified by a comma-separated list.
locations	Specifies a set of bounding boxes to track.
delimited	Specifies whether messages should be length-delimited.
stall_warnings	Specifies whether stall warnings should be delivered.

Table 8.1.: Parameters for the "POST statuses/filter" endpoint

8.1.1. Public streams

Public streams are the most suitable streams for doing some global analysis in real-time on Twitter. Public streams are divided in three endpoints :

- POST statuses/filter
- GET statuses/sample
- GET statuses/firehose

The "POST statuses/filter" endpoint returns statuses that match one or more filter predicates. Multiple parameters can be specified for filtering the stream as shown in the Table 8.1. At least one predicate parameter (follow, track, locations) must be specified.

The resource URL for accessing this endpoint is the following :

https://stream.twitter.com/1.1/statuses/filter.json

And parameters can be added, for example : track=foo&user=1234.

Then the "GET statuses/sample" endpoint returns a small random sample of all public statuses. For example, this endpoint is good for having information about what are the trending topics of the moment. The resource URL is :

https://stream.twitter.com/1.1/statuses/sample.json

Finally, the firehose endpoint returns all public statuses. However this endpoint requires special permission to access.

8.1.2. Data format

If nothing is specified, the data format returned is JSON. The Listing 8.1 is an example of a status update (a tweet) encoded in JSON.

The data contains information about the tweet : date of creation, text of the tweet, location from where the tweet has been posted, eventual hashtags, urls or user mentions,

etc. If the tweet is a retweet, information about the original tweet is also present. Besides information about the tweet, data also contains information about the user who posted the tweet.

8.1.3. Rate limit

The firehose is a source of all public streamed tweets. Typical accounts have access to around 1% of its totality. When using the filter endpoint, if the overall amount of matching tweets exceeds this limit of 1%, a rate limit notice telling how many tweets were not served will be display. It is difficult to increase this rate limit for the Twitter API; the best choice will be to look at other providers of Twitter data.

8.1.4. Other providers of Twitter data

The Twitter API is good if our needs don't exceed the rate limit. Otherwise, it will be better to choose an other provider. As we will see in our use cases, we will need much more than 1% of the whole stream. GNIP[1] is the largest provider of social data and is a partner of Twitter since 2010. GNIP provides the full Twitter firehose and, for our use case, we got the whole Twitter data for june 2012. Although the syntax is slightly different from Twitter, the main attributes and information are the same.

The GNIP dataset is a set of compressed text files. The files are distributed across folders, one per day. The whole dataset is about 800GB of compressed text files. The main difficulty with this dataset is that Storm is designed to process streams and not a finished set of data. To bypass this problem, we will simulate a stream by reading the files line by line and pushing them to a messaging queue system named Kafka. All the details will be described in chapter 10.

8.2. Bitly

Bitly.com is a very well known link shortener. It is mainly used for shortening links, especially on Twitter. For example, if someone wants to post a tweet containing a link to Wikipedia page of Albert Einstein, he can shortenize the link `http://en.wikipedia.org/wiki/Albert_Einstein` with Bitly. The final URL will be `http://bit.ly/4za3r`. In this url, "4za3r" is the code used by Bitly to match the shortened URL with the original one. In this report, we will simply call this code the Bitly code.

Bitly is hosted by Verisign and this is why this company can access the whole bitly stream. The bitly stream is actually separated in two streams : bitly encode and bitly

[1] http://www.gnip.com

```
{
"created_at":"Sat Apr 13 16:15:44 +0000 2013",
"id":323107241500753920,
"id_str":"323107241500753920",
"text":"My car needs a shower asap, thank you dust storm for that :)",
"source":"\u003ca href=\"http://twitter.com/download/iphone\" rel=\"nofollow\"\
    u003eTwitter for iPhone\u003c/a\u003e",
"truncated":false,
"in_reply_to_status_id":null,
"in_reply_to_status_id_str":null,
"in_reply_to_user_id":null,
"in_reply_to_user_id_str":null,
"in_reply_to_screen_name":null,
"user":{
  "id":112814547,
  "id_str":"112814547",
  "name":"Farah",
  "screen_name":"Farah327",
  "location":"www.changebyfas.com",
  "url":null,
  "description":" Designer, engineer-to-be, entrepreneur.\nInstagram: @changebyfas &
      @thedesignerkw ",
  "protected":false,
  "followers_count":644,
  "friends_count":592,
  "listed_count":1,
  "created_at":"Tue Feb 09 20:05:32 +0000 2010",
  "favourites_count":14,
  "utc_offset":-18000,
  "time_zone":"Quito",
  "geo_enabled":false,
  "verified":false,
  "statuses_count":25940,
  "lang":"en",
  "contributors_enabled":false,
  "is_translator":false,
  "profile_background_color":"C0DEED",
  "profile_background_image_url":"http://a0.twimg.com/images/themes/theme1/bg.png",
  "profile_background_image_url_https":"https://si0.twimg.com/images/themes/theme1/bg.
      png",
  "profile_background_tile":false,
  "profile_image_url":"http://a0.twimg.com/profile_images/3477060504/88078
      ca8e010538a10717b907514e888_normal.jpeg",
  "profile_image_url_https":"https://si0.twimg.com/profile_images/3477060504/88078
      ca8e010538a10717b907514e888_normal.jpeg",
  "profile_banner_url":"https://si0.twimg.com/profile_banners/112814547/1358855945",
  "profile_link_color":"0084B4",
  "profile_sidebar_border_color":"C0DEED",
  "profile_sidebar_fill_color":"DDEEF6",
  "profile_text_color":"333333",
  "profile_use_background_image":true,
  "default_profile":true,
  "default_profile_image":false,
  "following":null,
  "follow_request_sent":null,
  "notifications":null
  },
"geo":null,
"coordinates":null,
"place":null,
"contributors":null,
"retweet_count":0,
"favorite_count":0,
"entities":{
"hashtags":[],
"urls":[],
"user_mentions":[]
},
"favorited":false,
"retweeted":false,
"filter_level":"medium",
"lang":"en"
}
```


Listing 8.1: A Tweet encoded in JSON

```
{
  "a": "Mozilla\/5.0 (Symbian\/3; Series60\/5.2 NokiaN8-00\/014.002; Profile\/MIDP-2.1
      Configuration\/CLDC-1.1 ) AppleWebKit\/525 (KHTML, like Gecko) Version\/3.0 BrowserNG
      \/7.2.8.10 3gpp-gba",
  "c": "CR",
  "nk": 1,
  "tz": "America\/Costa_Rica",
  "gr": "08",
  "g": "9aL4Ni",
  "h": "cbzia0",
  "k": "504ede68-002e2-02553-3d1cf10a",
  "l": "ovila",
  "al": "es;q=1.0,nl;q=0.5,en-us, en;q=0.5,fr-ca, fr;q=0.5,pt-br, pt;q=0.5",
  "hh": "bit.ly",
  "r": "direct",
  "u": "http:\/\/ovi.la\/public-1-3\/trktemp",
  "t": 1348463726,
  "hc": 1270420113,
  "cy": "San Jos ",
  "ll": [ 9.933300, -84.083298 ],
  "i": "92f1f55de2ca9383ce827ed65324ee1d"
}
```


Listing 8.2: A Bitly tuple in JSON

decode. The bitly encode stream contains every new short link created and the bitly decode stream contains all the clicks on the shortened links.

In our use cases, we will mainly use the bitly decode stream. For a better understanding, the bitly decode stream will be called "bitly clicks" stream in this report. The Bitly decode is encoded in JSON and, as shown in the Listing 8.2, several attributes are present. For instance, there are information about the browser used, the Bitly code, the timestamp when the user clicked on the link and several geolocation information.

For our use cases we got the whole Bitly stream for june 2012 from VeriSign . This dataset consists of compressed text files and folders. The files are distributed in directories, one per day. The directories contain, generally one file per hour, so 24 files per directory. One month of Bitly stream represents about 30GB of compressed text files.

The compression used for the Bitly and Twitter (GNIP) files is LZO (Lempel-Ziv-Oberhumer).

8.3. Link between Twitter and Bitly

Bitly and Twitter are often used together. A tweet is limited to 140 characters. Therefore, when we want to post a tweet containing a link, the URL should be as short as possible. A good solution would be to first shorten the link with Bitly, and then post the shortened URL. Even if Twitter uses now its own link shortener on twitter.com, several applications for posting tweets still use Bitly as link shortener.

Having both Bitly and Twitter streams can be really interesting for doing different analysis. For instance, it will be possible to get tweets that contain a Bitly link and then

establish the timeline of the tweet, retweets and clicks. Having this information, it will also be possible to provide some advices on when and how to post a tweet.

9 Bitly Topology

9.1. Description of this use case 59
9.2. Spout implementation 62
9.3. Values extraction and blacklist check 63
9.4. SLD/TLD count and get webpage title 63

This chapter explains the development of a specific topology. Each component of a topology will be described.

9.1. Description of this use case

With this topology, we only used the bitly clicks stream to do some simple computation. This use case is a very good example to explore what can be done with Storm, test different possibilities, discover the possible issues and understand the main principles. Moreover, some bolts could be reused for other use cases.

As often when starting with a new tool, the first goal is to be able to run the project in local without any errors. Storm gives you the possibility to test your topology in local very easily. Before describing how to run a topology in local, let's analyze what should be present in our new topology. As seen in chapter 7 a topology is composed of spouts and bolts. Each spout and each bolt is declared in a specific java class and, therefore, in a specific file. A good practice would be to group every bolt and every spout under a specific package.

In addition to spouts and bolts classes, a main method must be present in order to be able to run the topology. This main method is responsible for :

- Setting all the configuration parameters. For example the number of workers, the maximum number of pending tuples, the tuple timeout, etc.

```
public class BitlyTopology {

  /*------- CONFIGURATION ----------*/
  private static final int timeToStore = 10; //seconds
  private static final int maxSpoutPending = 100000;
  private static final int numWorkers = 2;
  private static final int numAckers = 1;
  private static final int messageTimeout = 30; //seconds
  /* ---------------------------------- */

  private static LocalCluster cluster;

  public static void main(String[] args) throws InterruptedException, AlreadyAliveException,
      InvalidTopologyException {
    TopologyBuilder builder = new TopologyBuilder();

    builder.setSpout("inputStream", new HttpStream());

    builder.setBolt("getValues", new GetValuesFromJson(), 2).
      shuffleGrouping("inputStream");
    builder.setBolt("count", new CountingBolt(),2).
      fieldsGrouping("getValues", new Fields("user_id"));
    builder.setBolt("storage", new StoreToFile(),1).
      shuffleGrouping("count");

    /*------- SETUP CONFIG ----------*/
    Config conf = new Config();
    conf.setNumWorkers(numWorkers);
    conf.setMaxSpoutPending(maxSpoutPending);
    conf.setMessageTimeoutSecs(messageTimeout);
    conf.setDebug(true);
    /*-------------------------------*/

    /*------- LOCAL CLUSTER ----------*/
    cluster = new LocalCluster();
    conf.setDebug(false);
    cluster.submitTopology("BitlyTopology", conf, builder.createTopology());
    /*-------------------------------*/
  }
}
```

Listing 9.1: The main method of a Storm Topology

- Setting the different spouts and bolts, specifying which bolt subscribes to which spout(s) or bolt(s)
- Running the topology in local or in a cluster of machines

A simplified example of a main method of a topology is shown in Listing 9.1

From lines four to eight, we simply declare some configuration variables. TopologyBuilder in line 14 exposes the java API to specify a topology to be executed by Storm. Then, we define then our spout and bolts with the method setSpout or setBolt. Each Bolt must also subscribe to specific streams of another component (Bolt or Spout). Saying declarer.shuffleGrouping("getValues") means that this Bolt subscribes to the default stream of the bolt called "getValues".

It is possible for a bolt to emit more than one stream and also to subscribe to more than one stream. Consider the following example :

```
setBolt("BoltC", new BoltC(), 4)
  .shuffleGrouping("BoltA", "stream1")
```

```
3   .shuffleGrouping("BoltA","stream2")
    .fieldsGrouping("BoltB", new Fields("color"));
```

In this example, the BoltC subscribes to the stream1 and stream2 of BoltA and also to the default stream of BoltB. Due to shuffle grouping, the tuples from "BoltA" are randomly distributed across the tasks of "BoltC". The stream of BoltB is grouped by the field named "color". It means that tuples with the same "color" will always go to the same task.

Lines 26 to 30 in the Listing 9.1 are used to set the configuration parameters. The four following configurations are very common :

- TOPOLOGY_WORKERS (set with setNumWorkers) : "How many processes should be spawned around the cluster to execute this topology. Each process will execute some number of tasks as threads within them. This parameter should be used in conjunction with the parallelism hints on each component in the topology to tune the performance of a topology."
- TOPOLOGY_MAX_SPOUT_PENDING (set with setMaxSpoutPending) : "The maximum number of tuples that can be pending on a spout task at any given time. This config applies to individual tasks, not to spouts or topologies as a whole. A pending tuple is one that has been emitted from a spout but has not been acked or failed yet. Note that this config parameter has no effect for unreliable spouts that don't tag their tuples with a message id."
- TOPOLOGY_MESSAGE_TIMEOUT_SECS (set with setMessageTimeoutSecs) : "The maximum amount of time given to the topology to fully process a message emitted by a spout. If the message is not acked within this time frame, Storm will fail the message on the spout. Some spouts implementations will then replay the message at a later time."
- TOPOLOGY_DEBUG (set with setDebug) : "When set to true, Storm will log every message that's emitted." [Mara]

Storm has two modes of operations : local and distributed mode. For this use case, only the local mode will be used. In local mode, Storm simulates worker nodes with threads. This mode is really useful for developing and testing topologies. In Listing 9.1, the lines 34 to 36 are responsible for running the topology in local mode.

Now that we are able to run a simple topology in local mode, let's analyze the Figure 9.1 that represents the schema of this use case. Initially, we will go quickly through the topology and we will then analyze each part more deeply.

The start, as always in Storm, is the Spout, called "input Stream" in the Figure 9.1. The Spout simply reads a file line by line capturing through a streaming server. Then the values are extracted form the JSON line. Only a few fields are required for this use case, we will mainly use the final URL and the timestamp. An implementation of a bloom

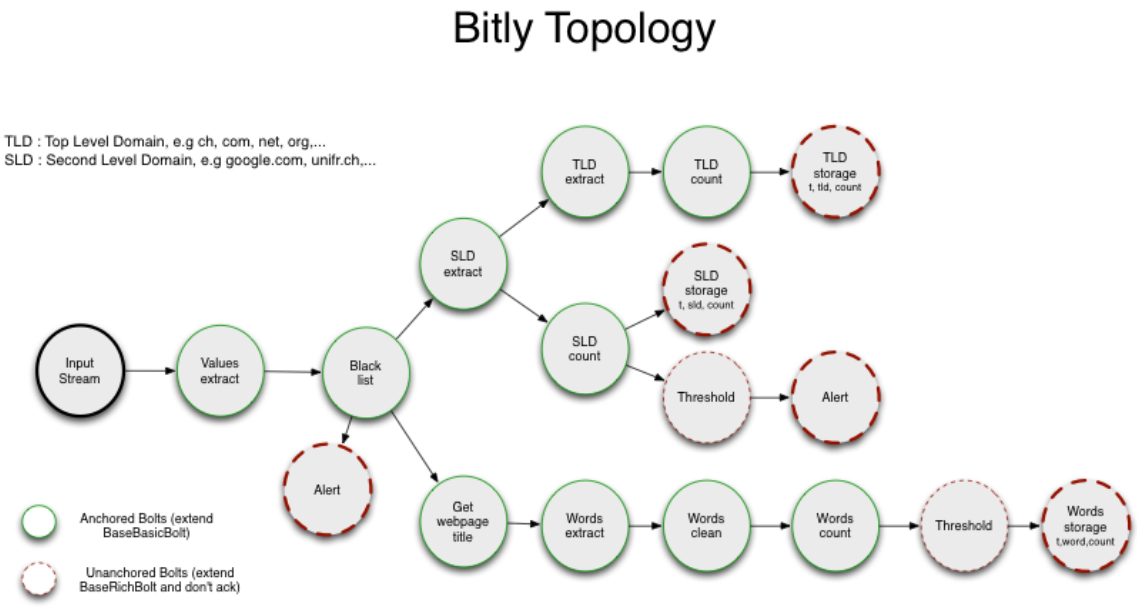


Figure 9.1.: Schema of the Bitly Topology

filter is used as a blacklist. Each URL is tested in order to know if it is blacklisted or not. The topology is then separated in two distinct branches. The first one counts the number of time distinct top and second level domain appears and regularly stores the results. The second branch tries to get the title of the webpage of the final URL, splits this URL, counts each word and regularly stores the results.

The following sections will precise each component of this topology.

9.2. Spout implementation

As said before, the spout is the source of streams in a topology. In this use case, the spout reads tuples from an external source and emits them into the topology.

The external source is a text file of 3GB streamed through HTTP with a Java application. The spout implementation is straightforward. There are three or five methods to implement; it depends on whether the spout is reliable or not. A reliable spout replays tuples that failed and an unreliable one doesn't. In our case, we chose to implement a reliable spout; therefore, the five methods are : open(), nextTuple(), declareOutputFields(), ack() and fail(). The implementation of this spout is shown in Listing 9.2.

The open method is responsible for connecting the external source to the topology (lines 39 to 53 in the Listing 9.2).

The most important method of the spout is nextTuple. Its function is to emit tuples into the topology. Since the spout is reliable, the failed tuples must be replayed. In order to

do this, we used two different hashmaps : "pendingTuples" and "toSend". When a tuple is read, it is put into both hashmaps (lines 19 and 20). Each iteration of the nextTuple method, all values in the toSend hashmap are emitted (lines 26 to 28) and the hashmap is cleared (line 29). The tuples can either succeed or failed. If a tuple is successfully completed through the topology, the ack() method of the spout is called and removes the tuple from the SpoutPendingTuples hashmap (line 56). In case a tuple failed, the fail() method is called, it retrieves the tuple from the SpoutPendingTuples hashmap and puts it again into the toSend hashmap in order to be emitted.

The last method to implement is declareOutputFields(). It simply declares the name of the fields present in the tuples. In our case, only one string field containing the JSON line is emitted.

9.3. Values extraction and blacklist check

After the tuples have been emitted into the topology, but before being used and analyzed, the tuples go through two different bolts : "Values extract" and "Blacklist" as shown in Figure 9.1.

The bolt "Values extract" parses the JSON line, keeps some key value pairs and emit them. In this topology we didn't need a lot of information; the URL of the link and a timestamp.

Then, there is an implementation of a Bloomfilter that tests if the URL is in a blacklist or not. A Bloomfilter is used to test whether an element is member of a set or not. Its main goal is to speed the process and avoid to always ask the disk. The blacklist is loaded into the Bloomfilter in memory in the "prepare" method of the bolt. A Bloomfilter has two configuration parameters : the number of expected entries and the percentage of false positive. When we ask a Bloomfilter to know if a element is member of a set and the response is "no", we have the guarantee that the element in not in the set. But if the response is "yes", it can be a false positive. Therefore, when the response is positive, it is recommended to check on the disk if the element is really present. The Figure 9.2 shows this notion of false positive.

It was interesting from the point of view of VeriSign (who handles the domain in .com and .net) to be able to check in real-time if a domain is blacklisted.

9.4. SLD/TLD count and get webpage title

After the bolt "Blacklist", the topology is divided into two different branches. The branch on the top is responsible for counting the second level domain (SLD) e.g "google.com",

```
public class HttpStream extends BaseRichSpout{

  private BufferedReader reader;
  private SpoutOutputCollector _collector;

  private final String URL = "http://minux.web4nuts.com:55555/decode_2012-09-24_06-15.txt?1";

  Map<Integer, String> SpoutPendingTuples;
  Map<Integer, String> toSend;

  int i=1;  //using for msgID

  @Override
  public void nextTuple() {
    try {
      String in;
      in = this.reader.readLine();
      if(in!=null){
          SpoutPendingTuples.put(i, in);
          toSend.put(i, in);
      }
      else {
          System.out.println("Nothing to read, wait 10 sec.");
          Thread.sleep(1000);
      }
      for(Map.Entry<Integer, String> entry : toSend.entrySet()){
        _collector.emit(new Values(entry.getValue()),entry.getKey()); //emit all the tuples
            from the toSend HashMap
      }
      toSend.clear(); //clear the HashMap
    } catch (Exception e){
      System.err.println("Error reading line "+e.getMessage());
      }
    i += 1;
  }

  @Override
  public void open(Map conf, TopologyContext context,
      SpoutOutputCollector collector) {
    this._collector = collector;

    this.SpoutPendingTuples = new HashMap<Integer, String>();
    this.toSend = new HashMap<Integer, String>();

    try {
      URL url = new URL(URL);
      this.reader = new BufferedReader(new InputStreamReader(url.openStream()));
    }
    catch (IOException e){
      System.err.println("Error in communication with server ["+e.getMessage()+"]");
    }
  }

  @Override
    public void ack(Object id) {
    //System.out.println("OK ack for "+id+" Size : "+SpoutPendingTuples.size());
    SpoutPendingTuples.remove(id); //Remove the tuple acked from the Map
    }

    @Override
    public void fail(Object id) {
      Integer msgID = (Integer) id;
      toSend.put(msgID, SpoutPendingTuples.get(msgID)); //Resend the tuple that failed
      System.err.println(id+" failed");
    }

  @Override
  public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declare(new Fields("str"));
  }

}
```

Listing 9.2: Spout's implementation

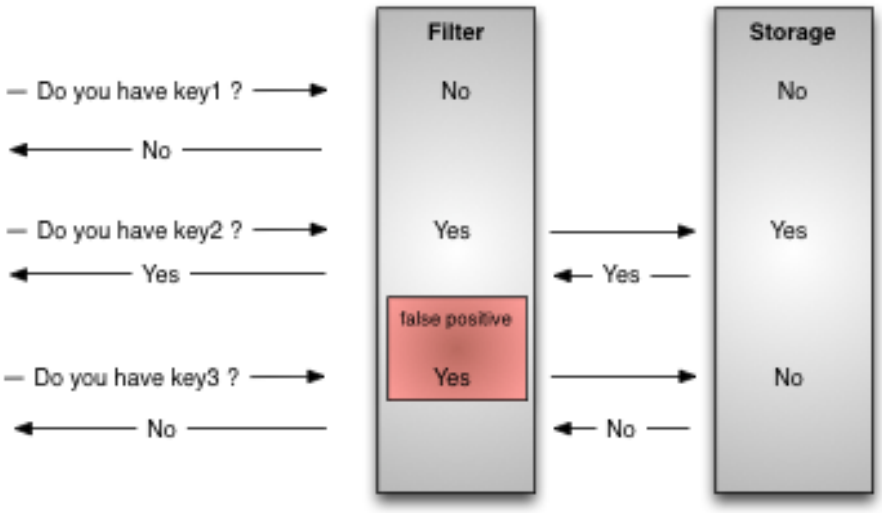


Figure 9.2.: Bloomfilter: false positive

"unifr.ch" and the top level domain (TLD) e.g ".ch", ".com". Then, every thirty minutes the results are stored in MySQL. The table in MySQL is very straightforward and has three fields : TLD/SLD, count, timestamp.

However, each tuple has a timeout. By default, it is thirty seconds. Therefore, if a tuple is still in the topology after thirty seconds, it will fail to be processed. In our case, the tuples wait until thirty minutes before being stored in the database. Consequently almost all the tuples in this topology will fail. In order to avoid this situation, the bolts after "SLD count", "TLD count" and "Word count" must be unanchored.

The notion of *anchoring* has been described in section 7.7. It means that the unanchored bolts (in red in the Figure 9.1) don't track the tuples. If a tuple failed to be processed in an unanchored bolt, it won't be considered as failed in the topology and won't be replayed.

Counting the number of occurrences of the SLD can be a useful use case for VeriSign, in order to detect, in real-time, possible attacks on domains.

The branch on the bottom tries to get the title of the webpage redirected by the bitly url. Then, a bolt splits the title word by word and another one cleans the words. The words are counted and emitted every thirty minutes. If the number of occurrences for a word reaches a threshold, the word and its count are saved in MySQL. The goal was to design a really simple trend detection in the bitly links.

However, the bolt that tries to get the webpage title was definitely the bottleneck of this topology. First, let's analyze its implementation. In order to get the connection with the server, the library HttpClient is used. A timeout of 500 milliseconds is setup, so that if the server cannot be reached during this timeout, the connection is aborted and

nothing is emitted. On the other hand, if the server can be reached, a BufferedReader reads the 5000 first characters of the response. If a "title" tag is matched inside this 5000 characters, a tuple containing the string inside the tag is emitted.

Even if the implementation of the bolt tries to limit the latency with a timeout and reads only the first 5000 characters, the bolt is significantly slower than the others, and is consequently the bottleneck. In Storm and in stream processing in general, it is really important to be able to process data at a higher rate that new data arrival rate.

There are three main possibilities to handle a bottleneck :

1. Increase the parallelism hint of the bolt
2. Reduce the number of tuples in the topology with the maxSpoutPending parameter
3. Increase tuple's timeout

The first possibility is probably the most common, add threads in order to parallelize the process. However, if it is not enough, another possibility is to throttle the spout and the rate at which new data arrives with the configuration parameter called "setMaxSpoutPending". If the number of maxSpoutPending is set to one, only one tuple at a time will be processed in the topology. The last possibility is to increase the tuple's timeout with the configuration parameter called "setMessageTimeoutSec". However, this parameter should be handled with care because if the timeout is too long, out of memory exceptions can occur.

A good practice would be to use a queue messaging system like Kafka or Kestrel between the input stream and the topology. Therefore it will be possible to handle peak in the stream throughput. Such a system will be used in the next use case.

10

Twitly topology

10.1. Description 67
10.2. Pushing data into Kafka 68
10.3. Data processing and storage 70
10.4. Trends detection 71
10.5. Results 73

This second use case mixes data from Twitter and Bitly

10.1. Description

The Twitly topology uses both Twitter and Bitly streams. From the Twitter side, the goal is to process and store tweets that contain a Bitly link. From the Bitly side, the goal is to store links that are present in the Twitter stream.

The goal is then to analyze the timeline of the tweets, retweets and clicks. During this chapter the term "click" refers to a click on a bitly link present in a tweet.

Initially, the Twitter API was used for capturing the Twitter stream. The idea was to search for tweets that contain a Bitly link with the following endpoint : `https://stream.twitter.com/1.1/statuses/filter.json?track=bit,ly`.

Even if this option was working well, the number of tweets lost due to the rate limit was quite high. Therefore, it was impossible to be sure of the accuracy of the results. The solution was to use the full Twitter and Bitly stream of June 2012. This dataset was made available by VeriSign.

In parallel, an advanced process for trends detection based on hashtags was used in this use case.

The first step was to find a way to simulate a stream in order to process it with Storm. This process will be described in the next section.

10.2. Pushing data into Kafka

As already described, Storm is mainly used as a stream processing system that process data as it arrives, in real-time. The problem with this use case is that a finished dataset of 1600GB is not a stream. At first glance, we should rather used Hadoop in order to process this dataset.

The first question was how to access the data throughout the cluster. A solution would have been to put the files in HDFS and, then, read them with a spout. However, the chosen solution was quite different. A script decompressed the files and read them line by line. Each line was put into to a queue messaging system named "Kafka". Then, a spout read the messages from Kafka and emitted them in the topology.

Apache Kafka is a high-throughput distributed publish-subscribe messaging system. It was first designed at LinkedIn. In July 2011, Kafka entered the Apache incubator with the goal of growing the Kafka community. It is designed to support the following [Foub]:

- Persistent messaging with O(1) disk structures that provide constant time performance even with many TB of stored messages.
- High-throughput: even with very modest hardware Kafka can support hundreds of thousands of messages per second.
- Explicit support for partitioning messages over Kafka servers and distributing consumption over a cluster of consumer machines while maintaining per-partition ordering semantics.

In this use case, two machines were used for Kafka. Actually, six machines were used in total but, generally, a machine shares its resources with different tools as described in the Figure 10.1.

It is important to notice that, since the version 0.8.0, Kafka enables replication across the brokers. However, the version of Kafka used in this use case was the 0.7.2. So this feature was not available. Another important notion in Kafka is those of partition. The number of partitions is configurable and can be set per topic and per broker. "First the stream is partitioned on the brokers into a set of distinct partitions. The semantic meaning of these partitions is left up to the producer and the producer specifies which partition a message belongs to. Within a partition messages are stored in the order in which they arrive at the broker, and will be given out to consumers in that same order." [Foub]

When using Kafka with Storm, an implementation of a KafkaSpout is available in the storm-contrib repository[1]. The maximum number of parallelism hint for a KafkaSpout is the number of partitions of the corresponding Kafka topic.

[1] https://github.com/nathanmarz/storm-contrib

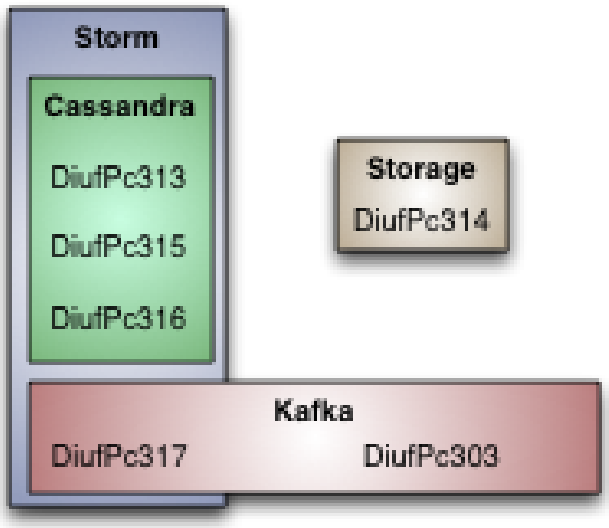


Figure 10.1.: Usage of the machines

The files were located on a NAS (Synology DS213+) mounted on a linux machine (Ubuntu 21.04 LTS). A script written in Java decompressed the file with the lzop[2] program and, then, the script read the file line by line and emitted each line to a Kafka topic (either Twitter or Bitly in this use case). Once the file had been fully read, the script deleted the file and the process restarted with the next file.

Of course, the files were read in a chronologic way. The files were present into directories, one for each day of the month. The name of the file contained the time of the first record. For example, the following file contains the twitter stream and starts the first of June at 01:20:51am :

```
1 gnip-201206/01/vscc-twitter-link-stream-20120601-012051-+0000-702275.txt.lzo
```

The file responsible for the Kafka configuration is present in "KafkaDir/config/server.properties". In this configuration file, some parameters must be setup:

- BrokerId : the id of the broker. This must be set to a unique integer for each broker.
- Hostname : the hostname that the broker will advertise to consumer.
- log.dir : the directory under which to store log files.
- num.partitions : the number of logical partitions per topic and per server. More partitions allow greater parallelism for consumption, but also mean more files.
- log.retention.hour : the minimum age of a log file to be eligible for deletion
- log.retention.size : a size-based retention policy for logs. Segments are pruned from the log as long as the remaining segments don't drop below log.retention.size.

[2]http://www.lzop.org

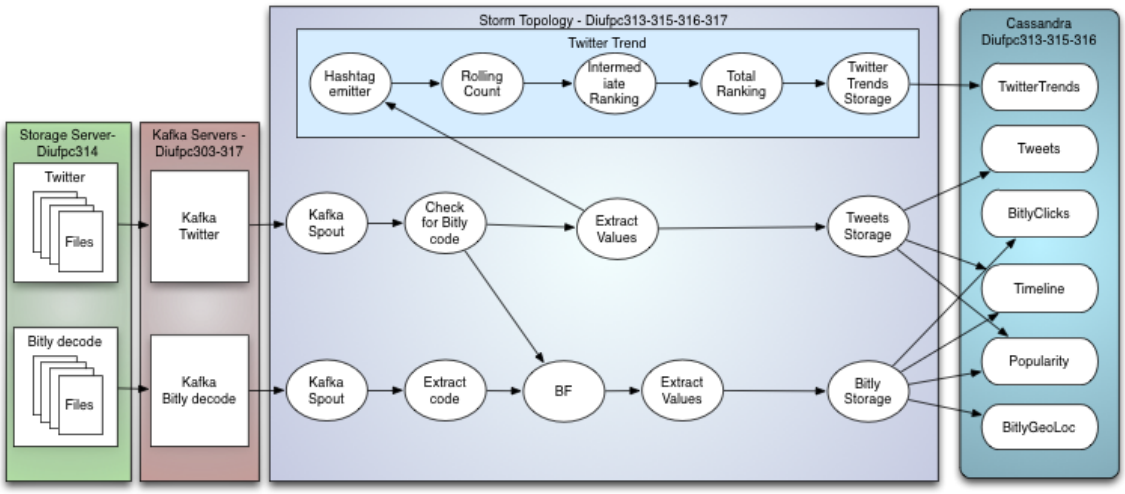


Figure 10.2.: Overview of the Twitly use case

- log.file.size : the maximum size of a log segment file. When this size is reached a new log segment will be created.

It is important to setup data retention correctly. Otherwise, the disks will be full within a few hours.

Once the configuration file is setup and if zookeeper is already started there is only one deamon to run for starting Kafka : kafka-server-start.sh . As always it is recommended to start this deamon under supervision. Therefore, if a problem occurs, the deamon is automatically restarted.

10.3. Data processing and storage

Once the data had been put in Kafka, a KafkaSpout read the messages and emitted them. The Figure 10.2 shows the topology. In the Twitter side, the first bolt checked if a Bitly link was present in the tweet. If there wasn't, nothing was emitted; if there was the bolt emitted two streams. One to the bolt called "BF" (BloomFilter) and one to the bolt that will parse the JSON line.

As in the Bitly topology of last chapter, an implementation of a bloom filter was used to store, in memory, the bitly code. Then, from the Bitly side, each bitly link was checked in order to know if the bitly code was present in the bloomfilter. However, in this use case, the number of expected elements was quite huge: approximately 300 millions entries. Consequently, it was quite hard to find a right configuration, with a reasonable percentage of false positive, without running into out of memory exception.

In order to replace the bloomfilter, different options were considered. First, as the information about the bitly code present in the tweets will be stored in Cassandra, it would have been possible to check, for each bitly link, directly in Cassandra if this link was present. This solution was working but was not the best. On average, 37'000 bitly links were read per second, so we should have queried the database 37'000 times per second, in order to know if the link was present. Furthermore, a lot of writes per second are also done in Cassandra. Even if Cassandra is quite good for reads and writes, the chosen solution was to use another key value store.

"Redis" is the key value store used for storing bitly code. Redis is an open source, BSD licensed, advanced key-value store. A lot of clients are available for almost all languages. Since this use case is coded in Java, the client used was Jedis[3], a java client for Redis.

The process was then quite simple. When a Tweet contained a Bitly link, the Bitly code was stored in Redis. Then for each Bitly link we checked if the Bitly code was present in Redis. It it was, information about the Bitly link was stored in Cassandra.

The outputs of the topology were stored in six different column family (CF) in Cassandra. The CF "TwitterTrends" stored the top 100 trends for a specific time span, "Tweets" simply contained information about tweets. "BitlyClicks" contained information about clicks on Bitly links and "Timeline" was storing the timeline of tweet, retweets and clicks for a specific Bitly code. "Popularity" stored, for a specific Bitly code, the number of tweets, retweets and clicks and finally "BitlyGeoLoc" contained information about the location from where people clicked on Bitly links.

10.4. Trends detection

The branch on the top of Figure 10.2 was used to detect trends in Twitter. First let's describe what is a trending topic. According to the wiki page of Twitter a trending topic is : "A word, phrase or topic that is tagged at a greater rate than other tags is said to be a trending topic. Trending topics become popular either through a concerted effort by users, or because of an event that prompts people to talk about one specific topic. These topics help Twitter and their users to understand what is happening in the world." [Wik13e]

A trending topic is generally mentioned within a specific time span. For example, the most popular topic for the last five minutes would probably be different than the most popular topic for the last two days. The goal is to use a process in which we can specify the time span within which we want to search for trending topics.

The detection of trending topics was based on hashtags. Therefore, the bolt "Hashtag emitter" was used to emit all hashtags contained in the tweet.

[3]https://github.com/xetorthio/jedis

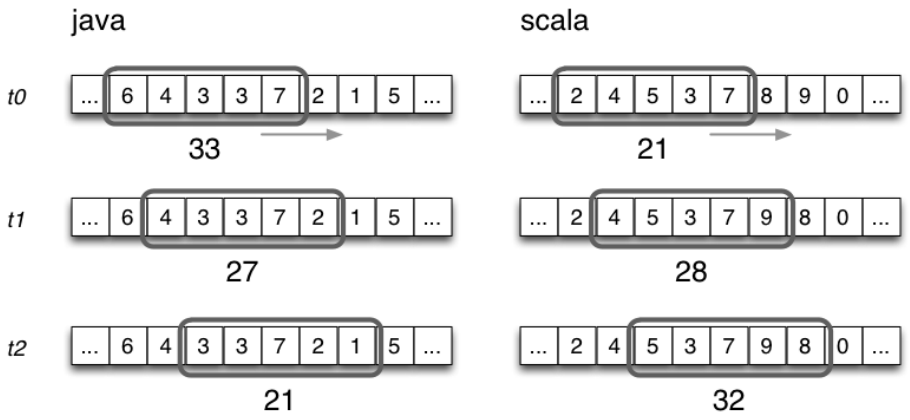


Figure 10.3.: Sliding Windows [Nol13]

In order to detect trending topics for a specific time span we used sliding windows, also known as rolling count. Figure 10.3 describes this notion of sliding windows. "As the sliding windows advances, the slice of its input data changes" [Nol13]. In Figure 10.3, the algorithm computes the sum of the current window's elements.

The sliding windows sum algorithm can be written as follows : $\sum_{i=t}^{i+m} element(i)$. Where t advances with the time and m represents the size of the windows [Nol13].

If the window advances with the time every N minutes, the elements in the input represent data collected over N minutes. Let's say that the window size is equivalent to Nxm where N=1 and m=5. Therefore, the sliding window algorithm emits the last five minutes aggregates every minute [Nol13].

Figure 10.4 shows how this algorithm is implemented.

1. First the bolt "hashtag emitter" emits all hashtag present in the tweets
2. Multiple instances of "Rolling count bolt" perform a rolling count of incoming hashtags
3. Multiple instances of intermediate rankings bolt (I.R bolt) distribute the load the load of pre-aggregating the various incoming rolling counts into intermediate rankings.
4. The final step is a single instance of Total rankings bolt which aggregates the incoming intermediate rankings into a global, consolidated total ranking. This bolt outputs the currently trending topics in the system [Nol13].

Of course, it is possible to specify the length of the sliding window and the emit frequencies. For example, it is possible to have a sliding windows of nine seconds and emit the last rolling count every two seconds or a sliding windows of five minutes and emit the last rolling count every minute.

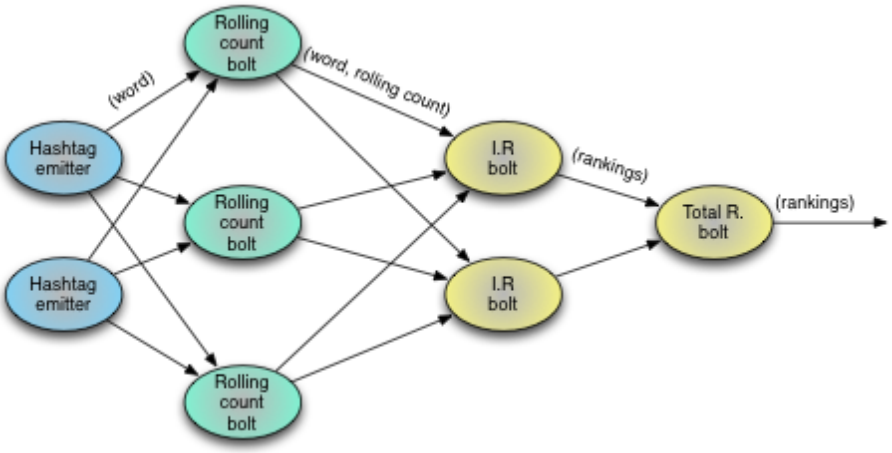


Figure 10.4.: Trends detection with a rolling count algorithm

In this use case, we chose a sliding window of five minutes and the emit frequency of the total rankings bolt was 20 seconds. Moreover only the first hundred trends were stored. In other words, every 20 seconds the top 100 trends for the last five minutes were stored.

10.5. Results

This section analyzes the results obtained with the topology. The main problem with the results is that it is impossible to know from which platform the link has been clicked. A Bitly link can first be shared on Facebook, then on LinkedIn or on another website before being posted on Twitter. Therefore, we cannot have the guarantee that the click on the link has been done from Twitter.

The expected result was to be able to have the timeline of a tweet, its retweet and the clicks on its bitly link. The ideal example would be to have first a tweet, a click on its bitly link then a retweet and other clicks and so on... However, in a lot of cases we have first clicks on the Bitly link and then a tweet. It means that the bitly link has already been used, maybe by other users and on other platforms before being posted on Twitter.

Another point is that there are also some applications (e.g for smartphones) that automatically post a tweet with a bitly link and generate and lot of data. Therefore the ideal cases, when a user first post a tweet with a bitly link and then there are clicks on the link are lost in the amount of data created by such applications and tools.

In order to really use and process the results of this topology, it would have been necessary to first try to "clean" the data and extract meaningful cases.

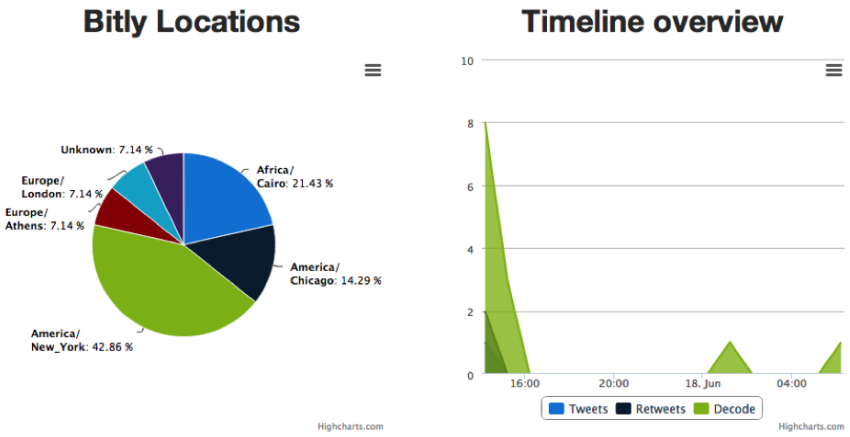


Figure 10.5.: Data visualization with HighCharts

However, Figure 10.5 shows a small web interface that uses a javascript library[4] in order to generate graphs. The graphs created are based on a Bitly code; for each Bitly code, we can have information about the different locations from where the link has been clicked and, an overview of the timeline of tweet(s) and retweet(s) that contain the bitly link and, of course, clicks on the link.

Figure 10.6 provides an other example of visualization using a nice tool called "Timeline JS"[5]. This tool provides an advanced timeline based on a JSON file that contains information about each (re)tweet and bitly click. Everytime the page is requested with a specific bitly code, a PHP script collect information contained in Cassandra and generate the JSON file with all the information. If the object is a tweet or a retweet, the tool displays the message of the tweet and, if the object is a click, the tool use the Google Maps API in order to display the location from where the click has been done.

[4]http://highcharts.com
[5]http://timeline.verite.co

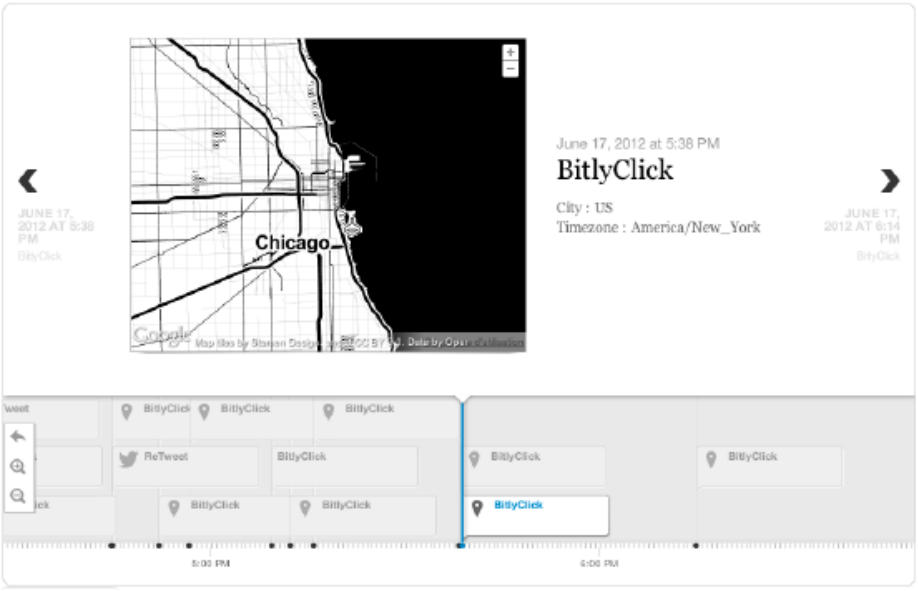


Figure 10.6.: Data visualization with Timeline JS

11

Performance and Storm Evaluation

11.1. Method employed 76
11.2. Results 77
11.3. Storm evaluation 78

This chapter describes the measured performance of Storm.

11.1. Method employed

On the Storm's website[1] it is said that Storm is very fast; "a benchmark clocked it at over a million tuples processed per second per node".

First of all there are different notions to analyze when speaking about performance. It is important to take into account the following :

- The kind of processing done : of course heavy computation will be processed slower than light computation.
- The hardware of the machines

Therefore, saying that it is possible to processed over a million tuples per second doesn't really make sense.

It was quite hard to measure the performance of the Twitly use case described in chapter 10. As data is stored only on one disk, it was the bottleneck. The performance of this use case is determined by the speed at which data can be read from the disk. However it was possible to process about 12'000 tweets per second and about 37'000 bitly clicks per second. This difference can be explained by the fact that a tweet encoded in JSON is about three time longer than a bitly click. Therefore for processing the whole dataset, 1600GB of compressed text file encoded in JSON, i.e one full month of Twitter and Bitly data, about eight days were necessary.

[1] http://www.storm-project.net

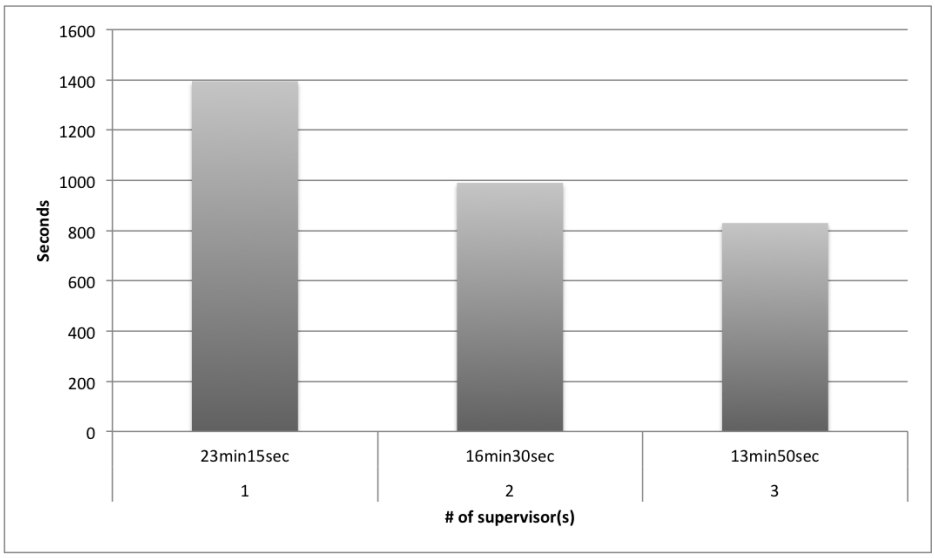


Figure 11.1.: Scalability

In order to analyze the scalability of Storm the following use case has been done : 34 GB of data (from both Bitly and Twitter) was loaded into Kafka. The processes done were quite similar to the Twitly use case without trends detection and storage in Cassandra. Therefore the topology was mainly parsing the data encoded in JSON.

The topology has been run three times with the same data and the same configuration; once with one supervisor, once with two supervisors and once with three supervisors. The goal was to analyze the performance based on the number of worker nodes.

11.2. Results

Two different observations have been done. First the time required for processing the 34 GB of data with respectively one, two and three supervisors and then the number of tuples processed according to the number of machines.

The Figure 11.1 shows the results based on the time required for processing the dataset. With only one supervisor (worker node), the 34 GB of data were processed in 23min15sec, i.e 24.9 MB/s. With two nodes, data was processed 405 seconds faster in 16min30sec (35.3 MB/s). Finally, with three nodes the process took 13min50 sec (41.9 MB/s).

We gained 405 seconds when passing from one to two supervisors and 160 seconds from two to three. Adding an additional node wouldn't be as useful as it seems that we almost reached the maximum capacity of the topology. With Storm and in stream processing in general, the maximum throughput is always dictated by the slowest entity.

In this experiment, the slowest entity was the bolt which parses Twitter's tuples. Another important point is that even if Kafka is an high-throughput messaging system it seems

that, with three supervisors we almost reached the maximum throughput possible with two Kafka brokers.

Instead of adding more machines for Storm, in order to upgrade the performance of this topology it would have been wiser to add more machines for Kafka.

11.3. Storm evaluation

In order to evaluate the different aspects of Storm, this section provides a SWOT (Strengths, Weaknesses, Opportunities, Threats) analysis of Storm. The Table 11.1 represents the SWOT matrix.

Let's start with the strengths. It is really easy to setup and run a topology, a lot of sample code are available, for example the storm-starter package that contains some topologies ready to get started. Moreover, if we encounter some problems, the community is large and responsive. Another good point is the permissive data model; a tuple is a named list of values and a value can be an object of any type. Storm has also a lot of advanced features, for example transactional topologies that guaranteed that tuples will processed exactly once. A lot of contribs are also available (Kafka, Kestrel, HDFS, Cassandra, ...). Finally Storm is already used in production by many companies which proves its stability.

Even if setup and run a topology is quite simple, finding the right configuration for the topology can be tricky. Actually the only way is to play with the different parameters and try to find out what is the best configuration. Another weakness of Storm is that it cannot dynamically scale when the throughput increases for example. It would be really nice if Storm would be able to detect bottlenecks and automatically resolved them. At the moment the only way for updating a topology is to stop it and restart it.

About the opportunities, Storm seems to be in the good way to become the standard in real-time data processing. It is currently used by many companies, it is free and open-source and the project is starred by more than 6000 users on Github. The market of stream processing is interesting because the demand is high but there is only a few concurrents.

Finally concerning the threats, even is there are some developers working on Storm, it seems that it is mainly a "one-man effort". The main developer is Nathan Marz[2] and he chose to develop Storm mainly in Clojure and Java. Clojure is a promising functional language inspired by Lisp. But it can be a barrier for developers who want to contribute to Storm but are not familiar with Clojure.

[2]http://www.nathanmarz.com

Strengths	**Weaknesses**
• Easy to setup • Large community • Very permissive data model • Features and contribs • Production ready	• Hard to find the right configuration • No dynamic scaling
Opportunities	**Threats**
• Become the standard in real-time data processing • Interesting market • Few concurrents	• One-man effort • Written in clojure

Table 11.1.: SWOT analysis of Storm

12 Conclusion

As described in the introduction, the main goals of this research were mainly focused on Storm and real-time data processing. Two use cases based on Storm were presented and the results were pretty good. It was shown not only how to build a new Storm project from scratch, but also how to use Storm in a complete environment including Cassandra, a NoSQL database good for both reads and writes; Redis, an advanced key-value store; Kafka, a high-throughput queue messaging system.

Storm is mainly used as a stream-processing engine. However, in our use cases, we mainly used a finished dataset and we simulated a stream by reading it. This solution was probably even more difficult than dealing directly with a big stream of data. Nevertheless, the solution presented was very convincing and it would have been really easy to adapt it in order to directly consume a stream.

With both Bitly and Twitter data, some interesting researches can be done. However, in this thesis, we were mainly focused on the process of the data. Future works, based on the output of this research could be done in order to deepen the correlation between Bitly and Twitter.

All in all, Storm seems to be a really promising tool for stream processing. It has a large community, nice and useful features, it is ready to be used in production and is actually used by many companies. Some improvements about dynamic scalability and update on the fly could be required. As shown in section 6.3, Storm can also be used in complement of Hadoop, in order to avoid the high latency updates of MapReduce.

A
Common Acronyms

API Application Programming Interface

ASCII American Standard Code for Information Interchange

CRM Customer Relationship Management

DBMS Database Management System

CQL Cassandra Query Language

CVS Concurrent Versions System

EJB Enterprise Java Beans

ERP Enterprise Ressource Planning

HDFS Hadoop Distributed File System

HTML Hypertext Markup Language

JSON JavaScript Object Notation

NoSQL Not Only SQL

RDBMS Relational Database Management System

SLD Second Level Domain

SQL Structured Query Language

TLD Top Level Domain

XML Extensible Markup Language

References

[Bro] Julian Browne. Brewer's cap theorem. http://www.julianbrowne.com/article/viewer/brewers-cap-theorem. 17

[Data] Datastax. Cassandra architecture. http://www.datastax.com/docs/1.2/cluster_architecture/index. 20, 21, 23

[Datb] Datastax. Introduction to composite columns. http://www.datastax.com/dev/blog/introduction-to-composite-columns-part-1. 23

[Din07] Doug Dineley. Truviso: New tricks with old sql. http://www.infoworld.com/t/hardware/truviso-new-tricks-old-sql-363, May 2007. 38

[Foua] The Apache Software Foundation. Apache drill. http://incubator.apache.org/drill/. 36

[Foub] The Apache Software Foundation. Apache kafka apache kafka: A high-throughput distributed messaging system. http://kafka.apache.org/. 68

[Fouc] The Apache Software Foundation. Hadoop documentation. http://hadoop.apache.org/docs/stable/. vii, 28

[Foud] The Apache Software Foundation. S4: distributed stream computing platform. http://incubator.apache.org/s4/. 38

[Hur] N. Hurst. Visual guide to nosql systems. vii, 18

[IBM] IBM. Infosphere streams. http://www-03.ibm.com/software/products/us/en/infosphere-streams. 37

[IDC11] IDC. Extracting value from chaos, june 2011. 9

[Ins11] McKinsey Global Institute. Big data: The next frontier for innovation, competition, and productivity. Paper, June 2011. 7, 9, 10, 11

[Lam11] C. Lam. *Hadoop in Action*. Manning, 2011. 26, 28

[Loh12] Steve Lohr. The age of big data. *The New York Times*, 2012. 6

[Mara] Nathan Marz. Class config: Java documentation. http://nathanmarz.github.io/storm/doc/backtype/storm/Config.html. 61

[Marb] Nathan Marz. Storm wiki. https://github.com/nathanmarz/storm/wiki. ix, 41, 46, 47, 48, 49, 50, 51

[Marc] Nathan Marz. Website of the storm project. http://www.storm-project.net. 42

[Mar11] Nathan Marz. A storm is coming: more details and plans for release. *Twitter Engineering*, 2011. 42

[Mar12] Nathan Marz. *Big Data - Principles and best practices of scalable realtime data systems*. Manning, 2012. 8, 14, 15, 39, 40

[Met] Cade Metz. Yahoo! tech boss gazes beyond hadoop. http://www.theregister.co.uk/2011/06/30/yahoo_hadoop_and_realtime/. 33

[MIT02] MIT. *Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services*, 2002. 17

[MK] Justin Erickson Marcel Kornacker. Cloudera impala: Real-time queries in apache hadoop, for real. http://blog.cloudera.com/blog/2012/10/cloudera-impala-real-time-queries-in-apache-hadoop-for-real/. 37

[MS] Mu-Sigma. To get the right answers one must first ask the right questions. http://www.mu-sigma.com/analytics/ecosystem/dipp.html. 34

[Nol12] Michael Noll. Understanding the parallelism of a storm topology. http://www.michael-noll.com/blog/2012/10/16/understanding-the-parallelism-of-a-storm-topology/, May 2012. vii, 44, 45

[Nol13] Michael Noll. Implementing real-time trending topics with a distributed rolling count algorithm in storm. http://www.michael-noll.com/blog/2013/01/18/implementing-real-time-trending-topics-in-storm/, Januar 2013. viii, 72

[Per13a] Benoît Perroud. Apache cassandra. http://fr.slideshare.net/benoitperroud/cassandra-talk-jug-lausanne-20120614, 2013. iii, 13, 19

[Per13b] Benoît Perroud. A hybrid approach to enabling real-time queries to end-users. *Software Developer's Journal*, 2013. 33, 40

[PL03] Hal R. Varian Peter Lyman. How much information? 2003. Technical report, Berkeley, 2003. 9

[Str] StreamBase. Streambase, complex event processing, event stream processing, streambase streamin platform. http://www.streambase.com. 37

[TC09] Peter Alvaro Tyson Condie, Neil Conway. Mapreduce online. Technical report, UC Berkely, Yahoo! Research, 2009. 35

[Twi] Twitter. The streaming apis. https://dev.twitter.com/docs/streaming-apis. 53

[Whi12] Tom White. *Hadoop: The definitive guide*. O'Reilly, 2012. 7

[Wik13a] Wikipedia. Cap theorem. http://en.wikipedia.org/wiki/CAP_theorem, 2013. 16

[Wik13b] Wikipedia. Fault-tolerant system. http://en.wikipedia.org/wiki/Fault-tolerant_system, 2013. 14, 45

[Wik13c] Wikipedia. Ibm infosphere. http://en.wikipedia.org/wiki/IBM_InfoSphere, 2013. 37

[Wik13d] Wikipedia. Truviso. http://en.wikipedia.org/wiki/Truviso, 2013. 38

[Wik13e] Wikipedia. Twitter. http://en.wikipedia.org/wiki/Twitter, 2013. 71

Index

Big Data, 6

Cassandra, 19

Hadoop, 26
HDFS, 27

Lambda Architecture, 39

MapReduce, 28

NoSQL, 13

Performance, 76

Real-Time, 32

Storm, 41

Twitter API, 53