UNIVERSITÉ LIBRE DE BRUXELLES

INFOH-515 - BIG DATA

BIG DATA MANAGEMENT & ANALYTICS

Project BDMA - Phase 1

Authors:
Anass Denguir
Keneth Ubeda

Professor:
Stijn Vansummeren

May 3, 2019

Contents

Chapter 1

Introduction

The amount of data humans are generating nowadays is really huge. In 2005 a study showed that humans generated 150 exabytes (billion gigabytes) of data while in 2010 this number increased to 1200 exabytes, 80 times more in just five years [source: The economist, February 2010]. This exponential growth in the amount of data has generated traditional systems for managing data to evolve and thus be able to manage big data as efficiently as possible. In this work we will study some of the existing technologies to build a Big Data Management & Analytic platform (BDMA) by means of implementing such a platform for a smart city.

1.1 Internet of Things IoT

A well known field that is characterized for generating huge amount of data is Internet of Things (IoT). IoT consists of having almost any device connected into a network, most commonly to the internet, with the objective of transmitting any useful information. One common scenario is having sensors that send their measurements through the internet. Usually these measurements are sent to a central unit of processing (i.e. server in the cloud) that organizes and gives meaning to the data that is arriving. The devices (things) that conform the network (connected to internet) are usually very simple devices with very specific objectives. For example a temperature sensor whose only work is to measure the environmental temperature every 30 seconds. The fact that these devices are very simple, make them very reproducible and portable as thus we can find or allocate them almost everywhere. However, that fact also causes that all the processing of the data produced has to be done in a more powerful device. Depending on the amount of data generated and the velocity at which we want to process it, we could need a server, a group of servers up to a cluster of computers dedicated to receive and process data. The problem then is not only processing but storing and sending/receiving the data.

There are many applications within the field of IoT such as Smart home, Smart mobility, Connected Health , Smart farming, Smart cities among others. In this project we will focus in the case of a smart city for which we will design a Big Data Management system that takes into account some of the issues that IoT and Big data in general imply: Data Volume, Data Variety, Data Veracity and Velocity.

1.2 Case of study

The Brussels government is working to gain the title of smart city for what has started a project where it will equip a modest number of households and public buildings with sensors of different types. We will participate in this project as the designers and builders of a Big Data Management & Analytics (BDMA) platform that will store, manage and analyze this sensor data.

During the project, different kinds of sensors will be placed in different kind of spaces. A space basically represents a physical space such as an apartment, a house, a floor in a public building, etc. Each space is identified with an internal id (a number). Some Meta-data about each space is also stored, for example we store the municipality at which the space belongs, whether the space is private or public, the address of the spaces among other information (This will be explained with more details in chapter 3). We will work with 4 different types of sensors: temperature, humidity, light, and movement. One space can have many sensors of different types and each sensor is located under a coordinates system within the space (x and y, w.r.t. some "base coordinate"). We can see an example of a space in figure 1.1 that has 54 points and each point equipped with 3 sensors; temperature, humidity and light.

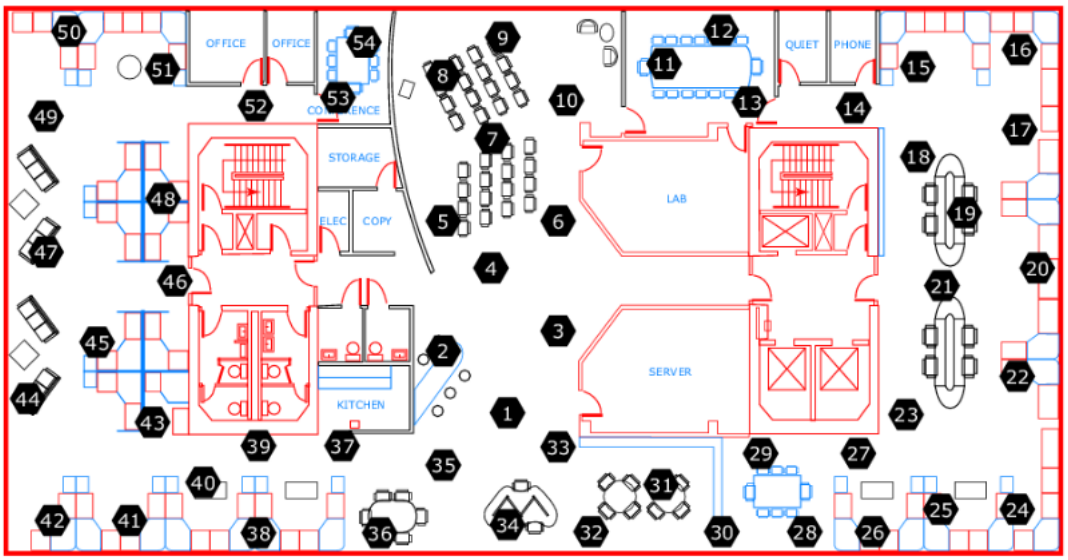

Figure 1.1: Space example

An additional information that is important to mention is the fact that the sensors are expected to send a measurements once every +/-30s; and initially, 10 000 spaces are participating to the project, each equipped with all 4 types of sensors. The project is divided into two phases: Phase I will consist in the setup of a Data pipeline and Phase II in applying Machine Learning methods on actual sensor data in order to build a model of the data that can be used for predictions.

1.2.1 Big Data Management Pipeline, Phase I

This report corresponds to the phase I of the project, and as first step of being able of managing such a smart city, we will design a Data pipeline that will be able of receiving data at a higher rate as well as being able of scaling in case the number of spaces increases.

We will explore some alternatives for the Big Data architecture (Chapter 2), we will go from traditional database systems to the λ-architecture and expose some of the problems we can face with traditional database systems and how the lambda architecture tries to solve them. Then we will move to the design and implementation (Chapter 3) of the BDMA Pipeline in which we will explain what tools we decided to use and the motivation behind using them. In Chapter 4 we will explain the queries we were asked to implement to show that the pipeline is working properly and we will also show the results of such queries in a Dashboard. We will finish this work by adding some conclusions after finishing the phase I of the project.

Chapter 2

Big Data architectures

In this chapter we will explore some alternatives that can be used to handle Big data, from storing it to processing it and to finally convert it in useful information. We will describe three different approaches, (i) Traditional database systems, (ii) the Lambda architecture, and (iii) the Kappa architecture. We will also describe which is the approach that better suits to our case and why.

2.1 Traditional database systems

Not long ago the most common solution to store data and querying it to perform some analyzes, was the Relational Database Management Systems (RDBMS). Some examples of such RDBMS are Oracle, Microsoft SQL, MySQL, PostgreSQL, among others. These systems provide a standard way of storing data as well as querying it and other mechanisms such as failure tolerance, replication, and most importantly a safe way of performing transactions while preserving the data integrity. In time these systems have gained the trust of many sectors in industry such as Banks, Insurances, Shops, e-commerce, etc. However, in the world of Big Data scaling is a key property that systems should have, so by means of an example we will try to explain how can we scale if using this type of database systems and observe if any problem is present.

2.1.1 Scaling with a traditional database

To explain how scaling works in a traditional database we will base on an example taken from the book Big Data, principles and best practices of scalable real-time data systems (see [1]).

Let us think that we were hired to create a web analytic application with the purpose of keep tracking of the number of visitors to any URL the customer wishes to track. One of the expected outcomes of this applications is to be able of telling you at any point in time the top 100 URLs with the major number of visits. So what we have to do if we use an RDBMS is to create the relational schema for the visits that will have: an **id**, the **url** to be tracked, and **user id**, and the counter of **visits** to keep track. So each time a page is visited the field visits will be incremented by one. An important

thing to mention is, that the RDBMS is hosted in a web server, so per each visit you have to ping the server to increment the visits value.

The problems start when your platform is getting a huge success and thus the traffic to the application is growing immeasurably. So if you would have a monitoring system, at some moment it will start warning about timeout errors on inserting the data into the data base. This can happen because at some point, when the amount of requests to increment the visits field is huge, the database can't keep the load. One solution to this problem could be to group wait for a number of visits to update the database not one by one visit but update 100 by 100 visits. Then it becomes more complex because it is needed to add a external tool, a queue for instance, between the server and the database to manage the grouping. This solution can work well and resolve the timeout issue.

We can think of have added a queue to manage the workload as putting a band-aid to the platform, because if the platform just keeps growing and growing, we should add workers to parallelize the updates, and that will just not help because the database is the bottleneck. Now things become more complex and a distributed technique has to be applied. One possibility is to use horizontal partitioning or sharding which consists in spreading the table across multiple servers and each server will have a subset of the data. What we can do to control what data goes to what server is using a mapping key system , in which you can have as much keys as servers. So we have to develop the mapping system that receive the data and in function of the key knows what data goes to which server.

The last solution, may solved the storage problem but at the same time is doing more difficult query the data to obtain the top 100 of more visited URLs. This is happening because now the data is spread across different severs, so now the platform should get first the top 100 for each shard and then merge those to give the global top 100. When the platform becomes more and more popular we will have to reshard the database into more shards to keep up with the write load. Then we to update our mapping system, if we forget to update this part, this can cause writings in the wrong shards and integrity and things can become really an irreversible disorder. Also having too many shards makes disk failures more likely to occur, and we will to configure fault-tolerance mechanisms such as master-slave replication or configure the queue in such a way that the update to shards that are not available stays in a separate pending status. This kind of configuration also allows human errors to permanently affect the data. For example a wrong code is the deployed to production that increments the visits by 2 for each URL instead of 1, we will not able to recover from that, even if we have a backup, because we cannot identify which data was corrupted.

2.1.2 Difficulties

Along this example we were identifying how can we scale in traditional systems and we have noticed some significant issues we could face when data is constantly grow-

ing (Big Data). It is clearly exemplified that a simple web analytics application became a very complex system with queues, shards, replicas, resharding, scripts, etc. One significant problem to notice is the fact that the system itself is not aware of its distributed nature. So it is impossible for the system to do anything about it, in other words it can not help to deal with the components of the distributed system, instead it needs external monitoring from experts to keep working correctly. It only becomes more an more complex, and therefore more prone to errors, when data is getting big and big. The conclusion here is that the traditional systems like relational database management systems (RDBMSs) are too complex for Big Data systems. We can think that, the solution is No SQL systems however, these systems can be in some ways simpler than traditional databases but being more complex in other cases. So the solution has to be further than just the database system, a new architecture is needed. A fundamentally concept to take into account in the architecture the incremental way of updating data, for that λ-architecture proposes a solution that will be explained in the next section.

2.2 λ-architecture

In order to circumvent the Big data management difficulties encountered by traditional database systems, we have to use an alternative architecture that respects the following constraints:

- The architecture has to be horizontally scalable. That means that the architecture has to scale by simply adding servers to the cluster.
- The architecture has to be robust and fault-tolerant. Indeed, on the one hand, the system has to tolerate server failures by using an appropriate distributed data architecture. On the other hand, the system has to be able to recover from human-related data corruptions.
- The architecture has to allow low latency reads and updates without losing its robustness.
- Finally, the data has to be rapidly processed through this architecture to extract value from it.

In order to respect these rules, the λ-architecture introduces a layer-based Big data system, which is divided into a Batch layer (see 2.2.1), a Serving layer (see 2.2.2) and a Speed layer (see 2.2.3). Each layer of the λ-architecture implements different features that fulfill some of the requirements mentioned above. This architecture is depicted in Figure 2.1. On the one hand, the data will be fed into the Batch layer, which will apply distributed computation to build multiple batch views. These views are stored in the Serving layer and are queried by the user to extract past information. On the other hand, data are streamed into the Speed layer to provide real-time views to the user. As it will be explained later, the Speed layer compensates the inherent latency induced by batch computing.

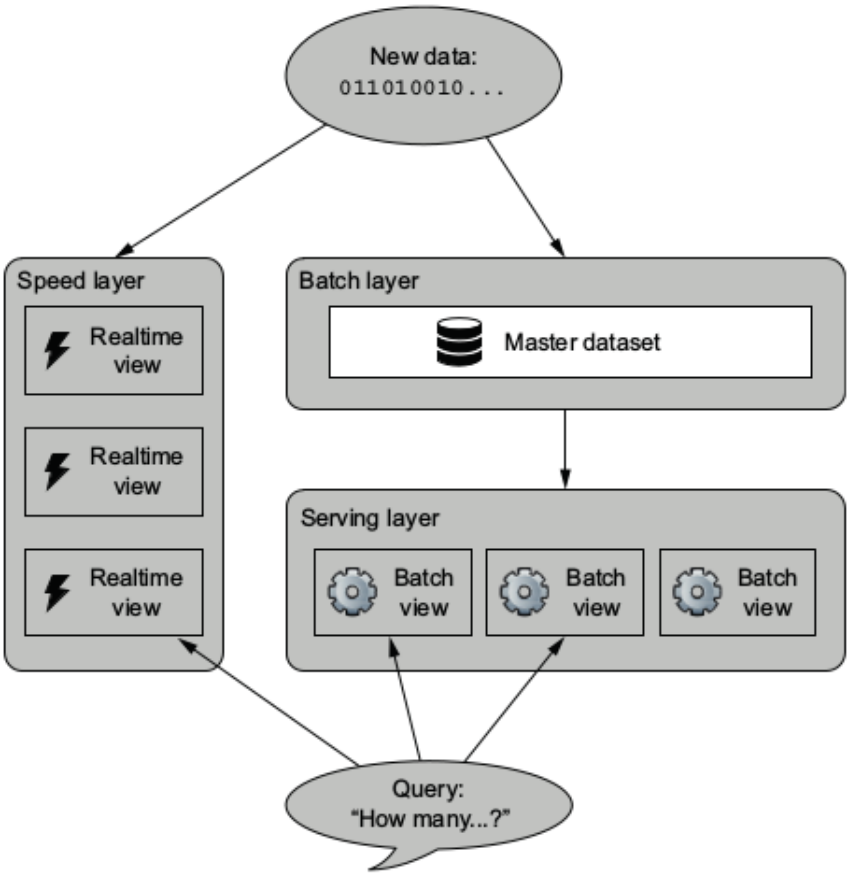

Figure 2.1: λ-architecture diagram

2.2.1 Batch Layer

The role of the Batch layer is twofold: (i) Storing the master dataset and (ii) Computing the batch views. The master dataset must store the raw data on which operations will be applied to extract valuable data. Storing the raw data is highly recommended since it maximizes the amount of information that could be derived and queried. Plus, it ensures that the original data can be recovered after being corrupted by applying some wrong operations on it. It is said that the master dataset is the source of truth. A good candidate for the master dataset is a Hadoop Distributed File System (HDFS) which natively tolerates machine failures by storing different copies of each file on multiple data nodes. Moreover, such a dataset prevents raw data corruption since it is an immutable data model. That means that no data can be lost, which ensures data recovery in case of data corruption.

Let us now discuss the structure and the format of the data. One can opt for either letting the data unstructured or normalizing it by means of joint tables. The choice of one of the two structures of data depends on the expected complexity of the operations required by the queries. If we are dealing with complex operations, keeping the data as raw as possible might be easier because the data can be fetched faster this way. Plus, the normalizing algorithms of data could eventually change later on. The

drawback of storing unstructured data is that it leads to potential duplication of data and consistency issues. Indeed, if any data is updated, one has to make sure that all the duplicated data are updated as well. This consistency problem is critical since the master dataset is supposed to be the source of truth. This issue will be investigated deeper in section 2.2.2. Finally, before storing the data in the master dataset, one has to decide on the expected format of the incoming data. The data can either be stored in a schema-less format (like *JSON* format) or be serialized to enforce it to follow a given schema (like in *SQL* databases). The latter solution is preferable as it avoids corrupted data to be stored in the master dataset. For example, if the *JSON* data presented below have to be stored in the master dataset, no error will be thrown during storage even if it clearly appears that the second line presents a wrong format for the field "age".

```
1 {"name": 'Bob', "age": 18}
2 {"name": 'Alice', "age": 'Hello'}
```

However, if the user queries the average of the age of people registered in the database, this corrupted data will give rise to an error during the processing of the data. This kind of error is very difficult to debug. Contrariwise, serializing data before storing has the advantage to throw an error during the writing of the data in the master dataset, which is more suitable to debug since the error is fully traceable. There are many serialization frameworks that can be used to perform this task such as Apache Thrift, Avro or Protocol Buffers.

The batch views contain the information that has been derived from the raw dataset. These are generated by applying distributed computation techniques on the master dataset. The data are processed batch per batch to optimize the throughput, but at the cost of a higher latency. These parallel computations are based on Map/Reduce architecture. There are many frameworks that implement distributed calculations, among which Hadoop and Spark.

2.2.2 Serving Layer

The Serving layer is responsible of storing the multiple batch views. In order to be scalable, the batch views have to be indexed and support distribution on multiple machines. The indexation method has to be designed to minimize the latency associated at each query. To do that, an indexation strategy is needed to minimize the amount of servers to fetch at each query. Indeed, the more servers we fetch, the higher the probability of having a server that will bottleneck the query. For instance, if the user emits a query composed of an URL and a given time interval, a huge delay will be induced if all the data corresponding to that URL are spread in different machines. The right indexing strategy consists of gathering all the data with the same URL in the same partitions and sorting these URL's by timestamp. This reduces the amount of disks to inspect and speeds up disk scans.

The split between the Batch and the Serving layers allows us to give an answer to the normalization problem introduced in section 2.2.1. The advantage of normaliza-

tion is that it ensures consistency of the data but at the cost of a longer latency for a given random query. However, the master dataset is not supposed to be randomly queried since the data are processed batch per batch. Hence, the data of the Batch layer can be normalized without suffering of extra latency. In contrary, the Serving layer can implement denormalization and perform some pre-computing on the data to speed up the query response because it is subject to random reads. This ensures a better availability of the batch views, but at the expense of a poorer consistency according to the CAP theorem. However, the inconsistency issues are less critical on the Serving layer than if it happened on the Batch layer since these can be corrected by simply recomputing the batch views.

One of the nice properties of the Serving layer database is that it does not require to support random writes, which are computationally expensive to handle. In summary, the database of the Serving layer has to meet the following requirements:

- be scalable and fault-tolerant: A distributed database is needed
- support batch writes to be updated by the master dataset
- support random reads to serve the user queries

In practice, any No-SQL database should be able to answer the above criteria, for example: Cassandra, ElephantDB, InfluxDB, etc.

2.2.3 Speed Layer

The architecture composed of the Batch layer and the Serving layer is enough to process, store and serve big data structures. However, such an architecture does not guarantee real-time views since batch computing involves high latency. The Speed layer is introduced to circumvent this problem. This layer is only responsible of processing recent data to generate real-time views. To do that, the Speed layer relies on incremental algorithms to update the state of the real-time views as data flow down. This ensures low-latency updates but at the expense of a poorer human fault-tolerance since the original data cannot be recovered by the Speed layer in the case of data corruption. However, any failure in the Speed layer is not so critical because the original data will be backed up by the Batch layer anyway. The Speed layer implements such a computation architecture because, in opposite to the Serving layer, its database does not only need to support random reads but also random writes, which are computationally expensive. Moreover, just as the Serving layer, the Speed layer has to tolerate machine failures and be scalable. Hence a distributed data storage system seems to be the appropriate solution. Any No-SQL database should meet these requirements.

An issue that has not been discussed yet concerns the data flow in the Speed layer. Indeed the database could either be updated synchronously or asynchronously. In the former case, the database is directly updated each time data is arriving to the Speed layer. This method has the lowest latency updates. However, a significant increase in the requests traffic can easily overload the database leading to timeout errors. To

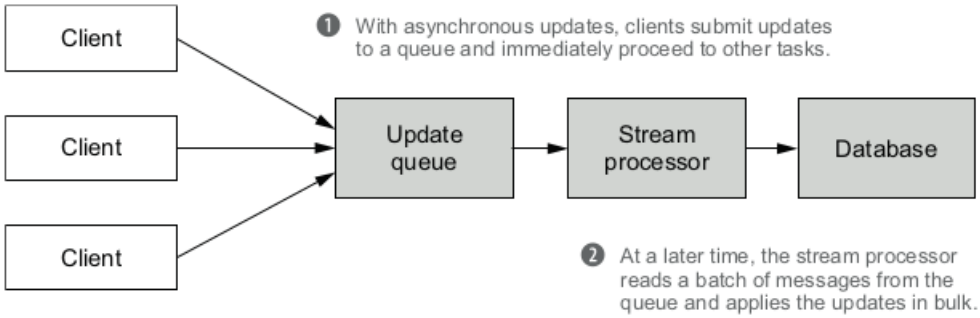

Figure 2.2: Asynchronous architecture for the Speed layer

solve this problem, an update queue can be integrated in the data flow. The latter buffers all the updates that have to be done on the database as depicted in Figure 2.2. This queue minimizes the risk of dropping a request, while allowing batch updates in the database. This asynchronous processing increases the throughput capacity at the expense of a higher latency. Furthermore, the queue has to support multiple consumers since many streaming applications will consume the data in parallel. To do it, the queue has to keep the data up to a certain amount of time after its consumption to guarantee all the consumers have processed it. Kafka is one of the most famous examples of multi-consumers queues. Once the data consumed, it is processed in a streaming fashion. Many streaming frameworks exist such as Spark Streaming, Storm, etc.

Finally, we have to manage the expiring of the real-time views. Indeed there is no need to store in the real-time views data that are available in the batch views. Hence, at the end of each Batch layer run, the data computed in the previous batch run can be thrown. But there is no easy way to distinguish data belonging to different batch runs. The simplest solution to this problem is to simply create two sets of real-time views as depicted in Figure 2.3. Each set accumulates and throws 2 runs of data in an alternative way. This ensures that the real-time data expires whenever these are written on the Serving layer, independently of the time required to process the batches.

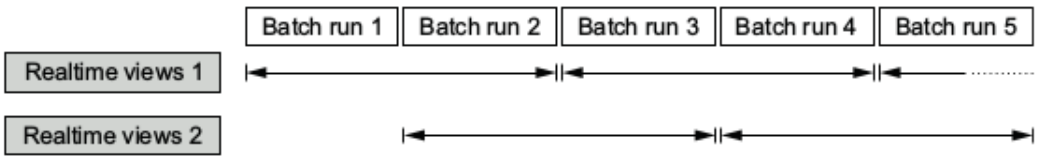

Figure 2.3: Complementary real-time views

Chapter 3

BDMA Pipeline

After having analyzed some existing architectures for Big data systems, we can now start describing some of the important characteristics of this specific project. We will first present a Data analysis that includes volume analysis and some Data assumptions we made. Then we will present our proposed design in which you will be able to have a good overview of the system as well as noticing which tools we used to build the system. To finish this chapter we will explain the details of the implementation, that includes an explanation of how each tool covers one part of the system and how they work as a whole.

3.1 Data analysis

To implement the BDMA pipeline, an analysis of the received data has to be done first. This analysis allowed us to learn some important features of the data we are dealing with. We consider that being aware of the data we are working with is a critical task for the design phase, because if we don't know well the data we cannot assure that the results are correct neither select the right tools to deal with it.

3.1.1 Volume analysis

In order to design the BDMA pipeline, it is important to have an idea of the expected data volume. This information will help to predict how much computation and storage resources will be needed to scale the BDMA. The raw data volume is simply proportional to the number of spaces S, the number of sensors per space N, the data rate of each sensor R and time t. The total volume is also proportional to the replication factor r, which ensures robustness against machine failures and inversely proportional to a compression factor c, if any compression algorithm is used. The expected data volume can thus be expressed by equation 3.1.

$$Volume = r \times \frac{1}{c} \times S \times N \times R \times t \tag{3.1}$$

For example, if we are dealing with $10,000$ spaces, each containing 4 sensors. Each sensors sends one measure every 30 seconds and the size of a single measure is approximately 50 bytes. If the data are stored with a replication of 3 and are uncompressed,

then the expected volume rate is expressed as follows:

$$VolumeRate = 3 \times 1 \times 10,000 \times 4 \times \frac{50}{30} \approx 12 \, \frac{MB}{min} \tag{3.2}$$

A direct consequence of this analysis is that the required data volume is growing over time and space. If we are only interested in accumulating one year of data, then the expected uncompressed volume of the master dataset is the following:

$$Volume = VolumeRate \times 1 \, year = 6.3 \, TB \tag{3.3}$$

However, to speed up the response of the queries, additional pre-computed queries are also stored in an HBASE database (we used OpenTSDB). Indeed, pre-aggregated data are generated by pre-filtering raw data in 3 different space granularities (space, municipality and city). For each space granularity, some statistical operations are also pre-computed (min, max, sum and $count$) on a $1minute$ time window before being stored in the database. These extra data considerably increase the expected data volume. Indeed, for the case of the city of Brussels, for every minute aggregation, the number of extra-data that will be generated is:

$$ExtraVolumeRate = r \times \frac{1}{c} \times 4 \times (S + 19 + 1) \times N \times 50 \, [Bytes/min] \tag{3.4}$$

Where:

- 4 is the number of statistical operations that are pre-computed per minute
- 19 corresponds to the number of municipalities in Brussels
- 1 is the number of cities (Brussels alone)
- 50 is the estimated size of one data (in bytes)

For our previous example, the number of spaces is $S = 10,000$, each containig $N = 4$ sensors. The replication factor remains $r = 3$. However, the compression factor of the database is around $c = 4.2$ according to OpenTSDB documentation. The extra volume rate is thus found by replacing these values in equation 3.4:

$$ExtraVolumeRate = 3 \times \frac{1}{4.2} \times 4 \times 10,020 \times 4 \times 50 = 5.7 \, \frac{MB}{min} \tag{3.5}$$

We can now correct the expected data volume of the previous example. On the one hand, the master dataset only contains the raw data stored in a HDFS. On the other hand, the HBASE database contains a replication of the raw data and the aggregated data, which amounts to a total data volume rate of:

$$TotalVolumeRate = (1 + \frac{1}{c}) \times VolumeRate + ExtraVolumeRate = 20.56 \, \frac{MB}{min} \tag{3.6}$$

For one year of data accumulation, the required data storage is thus 10.8 TB. This storage could be reduced further by compressing the master dataset. Concerning the computation of the data, one can see in equation 3.5 that the aggregation of data in spaces does not require huge RAM usage, unless the number of sensors are duplicated by a factor larger than 1000 with respect to the presented example.

3.1.2 Data assumptions

In this project, the sensors are simulated by a random data generator that generates data for each type of sensors in a specific data rate. Each data type has its own random data function which are designed to generate values that are not too far from what we could see in reality. Plus, knowing the distribution of the generated data is an advantage since we can easily verify the data integrity at the end of the pipeline by verifying if the data present the expected statistic distribution. The Table 3.1 lists all the statistical distribution generated for each sensor type. The data are generated as a String with the following format: $date\ time\ municipality - space - datatype\ value\ voltage$.

Sensor Type	Distribution	Parameters
Temperature	Gaussian	$\mu = 20, \sigma = 3$
Humidity	Uniform	$\min = 0, \max = 100$
Light	Gaussian	$\mu = 40, \sigma = 5$
Motion	Binary	$\min = 0, \max = 1$

Table 3.1: Statistical distributions of data generator

3.2 Design

For the advantages mentioned in Chapter 2 we decided to implement our solution based on the $\lambda-$architecture. So the architecture of the proposed solution is shown in Figure 3.1 and the set of tools used to implement it are listed in Table 3.2.

All the tools we have selected for the implementation of this project are able to scale up by working in a distributed environment. In Kafka we can add brokers to the cluster, Zookeper also works in a replicated mode to be fault tolerant, in Hadoop we can add data nodes, in Spark we add workers and OpenTSDB is distributed if the HDFS is. Also in Apache flume we can create as much agents as needed to increase the throughput of data. We know this is a very important characteristic to have within a system because as the system is distributed there is no single point of failure and the probabilities to fall down decreases with it. These set of tools are commonly used together which is an advantage for maintainability purposes and they are all well documented, which was fundamental for the development of this phase and will be for future improvements.

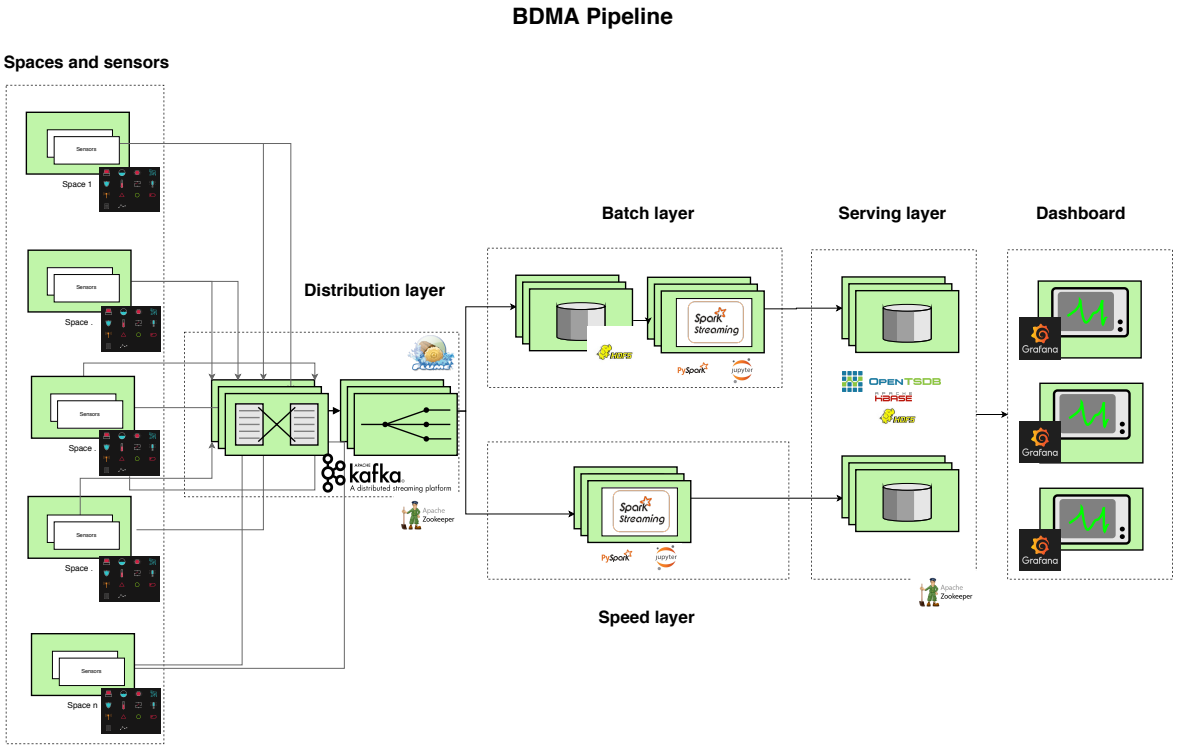

Figure 3.1: BDMA pipeline architecture diagram. Each dotted rectangle represent one layer of the architecture and the layers that work together are connected with a black bold line. If you see more than one green rectangle that encircle a tool, that represent the distributed behaviour of the tool.

Table 3.2: Set of tools used in the proposed solution

Tool	**Version**	**Purpose**
Apache Zookeeper	3.4.13	Kafka and OpenTSDB distributed coordination
Apache Kafka	2.2.0	Produce and consume sensor data
Apache Flume	1.9.0	Ingest data produced by Kafka into HDFS
Apache Hadoop	2.7.1	Distributed file system to store Master Data set and OpenTSDB data
Apache Spark	2.4.1	Process data in batch and streaming mode
PySpark	Python 3.7	Spark driver to work with spark in Python
OpenTSDB	2.4.0	Store data to perform queries that works on top of Apache HBase
Grafana	6.1.4	Visualization platform to show results of the queries

3.3 Implementation

In this section we will explain each of the components of the BDMA pipeline that we mentioned in the previous section (Design). We will go through all of them by describing their important characteristics and how they work in our set up environment. We will associate each of the tools with its corresponding layer in the λ-architecture to give already a hint of the tool's roll in the system.

It is also important to mention that we have implemented all the architecture utilizing Docker services. This allows to deploy and migrate our architecture to different environments in a very fast way as well as scale up services (Kafka, Hadoop, Spark, etc) by just changing parameters in a configuration file.

3.3.1 Distribution layer

This layer will be in charge of receiving and distributing the sensor data coming from the different spaces into their corresponding next step of the pipeline, batch layer or speed layer.

Apache Kafka

Apache Kafka is the application that directly communicates with the sensors in the BDMA Pipeline. Indeed, as explained in section 2.2.3, if the data generator (i.e the sensors) directly write to the database, the latter might raise a timeout error if the request traffic spikes. In order to deal with this issue, Kafka is used as an intermediate queue. As illustrated in Figure 3.2, this queue is categorized into topics in which the producers (the sensors) can submit their data without worrying about delivering it to the database. The Kafka brokers are responsible of partitioning the messages queued in each topic and storing them in a distributed fashion to guarantee scalability. Fault-tolerance is also ensured by replicating the queued data multiple times. All the distributed file system operaton are managed by Zookeeper, which is another another software on which Kafka depends. Kafka has the nice property of supporting multiple consumers. Indeed, since the brokers decide whether a data is old or not, the data are not destroyed when they are consumed but rather after a timeout decided by the broker. The consumers pull the messages from the topics they are subscribed to in order to process data when needed.

Apache Flume

Apache Flume as defined in its website is a distributed, reliable, and available service for efficiently collecting, aggregating, and moving large amounts of log data. It has a simple and flexible architecture based on streaming data flows. It is robust and fault tolerant with tun-able reliability mechanisms and many fail-over and recovery mechanisms. It uses a simple extensible data model that allows for online analytic application that we can see in Figure 3.3.

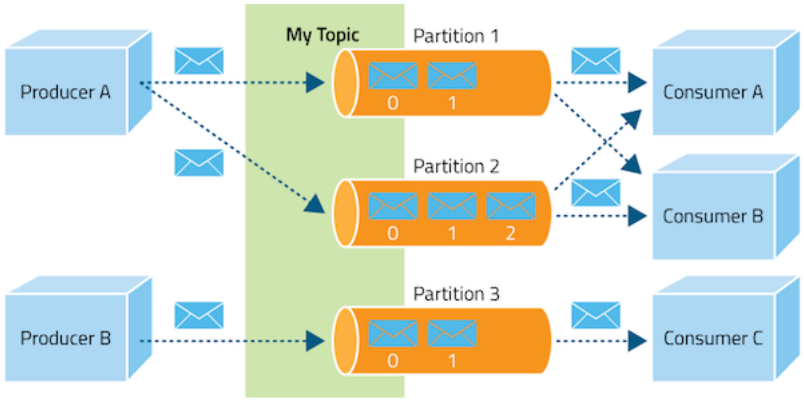

Figure 3.2: Kafka architecture

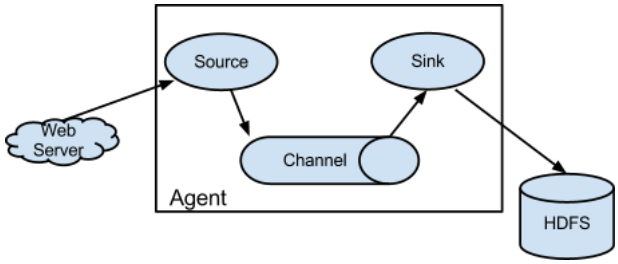

Figure 3.3: Apache flume model. As we can observe the main components of a flume agent are the **source**, **channel** and **sink**. For this implementation the source corresponds to a Kafka consumer, we use a memory channel and then the sink is HDFS.

By looking at the model and substituting the components of the agents it is pretty clear what flume is doing in the system: gathering data from a Kafka topic (source) through memory (channel) to finally write to HDFS (sink). The work of a flume agent is simple and specific and to implement it we have to use a properties file that contains information about the source, channel and sink such as the Kafka bootstrap servers, Kafka topics, type of channel the HDFS path to write the data, among others. We can create multiple agents, different properties files or the same but with different agent name, to increase the amount of data we can pass from one place to another in parallel.

3.3.2 Speed and Batch layer

As proposed in the λ-architecture, the Batch layer has two different roles. On the one hand, it stores data in the master dataset, which is a HDFS. On the other hand, batch data processing are applied on the master dataset to extract valuable data. However, as the majority of the batch processing mentioned in the query requirements implies gathering data per time and space granularities, we chose to handle only the space aggregations in the Batch Layer and let the Serving layer (OpenTSDB database) filter the data time-wisely since this database is specifically designed to handle time-series data. Therefore, the Batch layer can directly stream data from the master dataset and perform spatial aggregation on it. Eventually the Batch layer can roll-up aggregated data in different time windows to speed up queries involving long time windows. This sit-

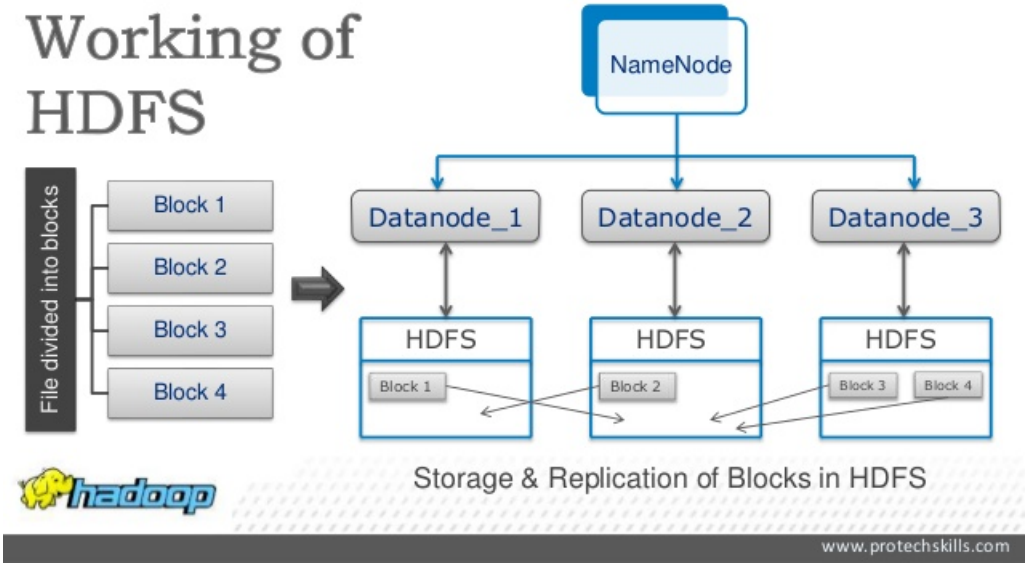

Figure 3.4: Working of HDFS

uation is depicted in Figure 3.1, where Spark Streaming is used to process data in both Batch and Speed layers. Besides, the Speed Layer processes data directly from Kafka, which implies a smaller latency than for the Batch layer. In this layer, the streaming data are pre-filtered per data type (e.g:temperature) in order to extract the most occurrent temperatures in a given time window. To do these queries, the Speed layer also uses Spark Streaming, which implies that the whole Pipeline is working in a streaming fashion. However the workload of the Speed layer is much lightweight than the one of its sibling layer. It is important to note that the fully streaming Pipeline design is a direct consequence of the choice of a time-series database for the Serving layer, which helps the Batch layer to perform computation in an efficient way.

Apache HDFS

Apache HDFS is distributed file system composed of two types of nodes: the namenode and the datanodes. The namenode knows where data are stored in the cluster, while the datanodes are simple data storage services. The working of HDFS is depicted in Figure 3.4. HDFS is used to store the master dataset of the BDMA Pipeline because of its following properties: Firstly, a HDFS is an immutable dataset, which prevents any data corruption. This is a nice property as the master dataset is considered as the source of truth of the BDMA Pipeline. Secondly, a HDFS is distributed, that means that data can be replicated on multiple data nodes. This prevents data loss in case of machine failure. Finally, the distributed nature of the HDFS makes it a ready-to-scale dataset, which is easily configurable in a Docker environment.

Apache Spark Streaming

Apache Spark Streaming is a distributed computation framework that transforms data streams from any service such as HDFS, Kafka or Twitter into a sequence of Resilient

Distributed Datasets (RDD), namely a DStream. Spark Streaming provides an abstraction functionality to treat an entire DStream as it were a single RDD. One of the nice properties of Spark Streaming is how easy it is to treat data belonging into a given time window, which is useful for most of the queries mentioned in the requirements. That is why, it is used in both the Speed and Batch layer of the BDMA Pipeline. Spark is used for processing the data because it is a distributed network of worker nodes managed by a master node. The scaling of the computation resources can thus be achieved by simply adding Spark workers in the network, which is particularly easy to do in a Docker environment.

3.3.3 Serving layer

The serving layer is an important part of the lambda architecture because it has to provide velocity and flexibility to querying the data. Usually in the serving layer we have lots of alternatives such as Cassandra, HBase, Neo4J; We could even have SQL databases since they are very useful to querying data. However the decision of choosing a database for the serving layer will highly depend on the type of data we are processing and the type of queries we want to perform from other platforms that might need the processed data. In our case, since we are working with Internet of Things we can think in a time series database for what we have some alternatives such as OpenTSDB, InfluxDB, Druid among others. We have chosen to use OpenTSDB because we found that it is highly compatible with the other technologies we are using (HDFS, Spark, Grafana, etc) . OpenTSDB also provides easy ways to scaling up and is optimized to receive thousands of points per second, which is one of the main characteristics data generated by IoT sensors. Another important feature that OpenTSDB has is the compatibility with visualization tools which in for this case we found Grafana (visualization framework) was the correct choice. We can easily connect OpenTSDB via its HTTP API as data source to Grafana and thus start querying data.

OpenTSDB

Let us first think why time series systems should have a different way of handling the data. We will point out to the differences between TSDB and traditional databases for what we categorize into two different levels: scale and usability.

Time-series databases (based on relational databases or NoSQL) handle scale by introducing performance improvements: higher insertion rates and better data compression, which is a vital feature when taking into account that time series data accumulates quickly.

The second factor is usability. This includes functions and operations that are common to time-series data analysis, including arithmetic expressions, string operations, aggregate functions, re-sampling into different time resolutions, ordering, ranking and limiting. Expressions and operators may refer to different series, e.g. sum up all time series whose tags match a specific pattern.

TSDBs can be categorized into four groups. The first group is composed of TSDBs dependent on an existing DBMS (HBase, Cassandra) to store the time series data.

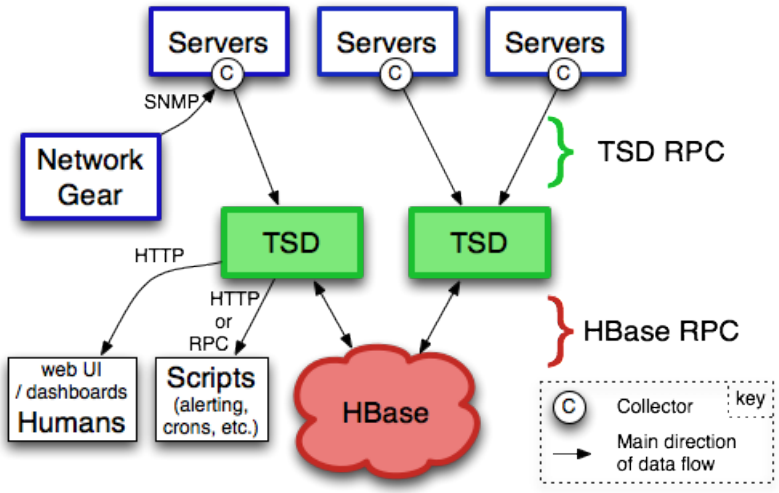

Figure 3.5: Opentsdb architecture. HBase it is connected to a Hadoop cluster and the TSDs will read data from spark streaming (servers in the picture) in our study case.

The second group covers TSDBs using DBMS for storing meta data. The third group comprises RDBMS. The last group contains all TSDBs that are not open source.

OpenTSDB lays in the category of "DBMS-dependent" and we can define it as a distributed and scalable time series database built on top of Hadoop and HBase as we can see in figure 3.5. The data schema is highly optimized for fast aggregations of similar time series to minimize storage space.

As in Relational databases we need to define a model for the data in OpenTSDB. We call such model a **Naming Schema** which is based on **metrics** and **tags**. In OpenTSDB each time series has a generic "metric" name, that can be shared by many unique time series. In the case study, we have 6 different metrics: **motion, light, humidity, temperature, temperature.occurrence, occurrence temperature**. Four of the metrics corresponds to the name of sensors which means that each of those metrics will contain the data corresponding to each sensor. For the temperature sensors we have two extra metrics that correspond will help us store the result of the second query (Chapter 4). What distinguishes a specific time series from another is the use of "tags", every time series must have at least one tag. We set 3 different tags: **city, municipality and space**, that can be used in the sensors type metric; for temperature.occurrence and occurrence.temperature only city applies .In order to observe this in a more understandable way you can see at table 3.3 with the metrics and tags that describes our data model.

OpenTSDB allows already to roll-up data in time and aggregate data by tags. This is a very nice feature that time series databases should have and OpenTSDB handle it pretty well. Rolling-up data could become a very hard task for OpenTSDB when users are trying to analyze big periods of time. fortunately since OpenTSDB 2.4 we can insert pre-rolled-up data and thus help OpenTSDB to serve queries as fast as possible. OpenTSDB is not only capable of use pre-rolled-up data but also pre-aggregated data which is useful when we data has a high cardinally (i.e. lots of measurements of temperature in different spaces).OpenTSDB can go even further and handle pre-

Metric	city	municipality	space
temperature	*Brussels*	*[1-19]*	*[1-54]*
humidity			
light			
motion			
temperature.occurence		*N/A*	*N/A*
ocurrence.temperature		*N/A*	*N/A*

Table 3.3: OpenTSDB Naming Schema (metrics and tags). This model is very easy to understand and we can easily identify the different levels of detail we can have regarding the cities, municipalities and spaces.

rolledup-aggregated data, which can dramatically change query times. It is important to mention that OpenTSDB does not pre-compute the data itself, instead it take advantages of streaming platforms that are optimized for that purpose as it is the case of Yahoo that uses Storm (Streaming framework) and OpenTSDB to monitor a massive scale of data.

We took advantage of spark streaming in this case to pre-compute some data and thus feed OpenTSDB with it (i.e. Max temperature aggregated by city). It is also important to mention that to insert data to OpenTSDB we post JSON data to the http API that OpenTSDB provides which makes very easy the transition of data from one place to another.

Grafana

The visualization part of this project cannot be neglected because if we do everything right (receive, store, process data efficiently), but we don't show the results properly they are meaningless. So we selected Grafana which is an Analytic platform that allows us to add OpenTSDB as data source to visualize and querying the data. This tool is specialized but not limited to time series analysis. So for this case in particular makes a good match, since it will help us to show the queries and monitor the data of all the sensors in all the different spaces in Brussels. With Grafana we are able to analyze time series data in batch and streaming mode because one of its features is the ability or receiving data as fast as possible.

In Grafana we can create Dashboards, save them and add different type of figures in them. We can already monitor the data in a window time (last 1 hour, last 24 hours, etc) which makes Grafana a very flexible tool since you can add a plot for each window of time you want. In Grafana we are also able of refreshing the query every certain time (1s, 5s, 1m, etc.). All these features make Grafana a flexible and efficient tool to visualize data that is not limited to only one data source, instead you can use different types of sources such as OpenTSDB, MSSQL, InfluxDB, Elastic Search, MySQL, among others. One last advantage of Grafana is the fact that it is a open source project which could be useful in case we want to develop more specific features.

Chapter 4

Analytics and results

In order to validate the well functioning of the design BDMA pipeline we were asked to show its performance through a set of queries. In this chapter we will explain what are the queries about, as well as showing their results in our Grafana dashboard.

4.1 Queries

The following, are the queries we will perform to validate our BDMA pipeline implementation:

1. Compute the *min*, *max* and *avg* per type of sensors (ie: temperature, humidity, light and motion sensors) for different time windows and space granularities.
2. Classify each 15 *minutes* timeslots into either day-time or night-time temperature, again for different time window and space granularities.
3. Provide an overview of the temperatures that are the most occurent in a 1h sliding window in the whole city of Brussels.

4.1.1 Statistics

Here are shown the results of computing basic statistics over the data.

Temperature sensors

In Grafana we can show the result of multiple queries in the same graph, so for this query The Max, Min and AVG values are shown in the same graph. Grafana also allows us to show data in a time-series style and for this particular type of sensor we chose to show 6 hours of data rolled up every 1 minute. We can clearly observe in the figures below that data behaves as the distribution that generates it. There are two types of graphs, the first one represents data aggregated by Brussels which means that this data comprises all the sensors in all spaces in Brussels, the operator used to aggregate was average. The second graph shows the same information as for the previous one, but now the data is aggregated into a municipality level.

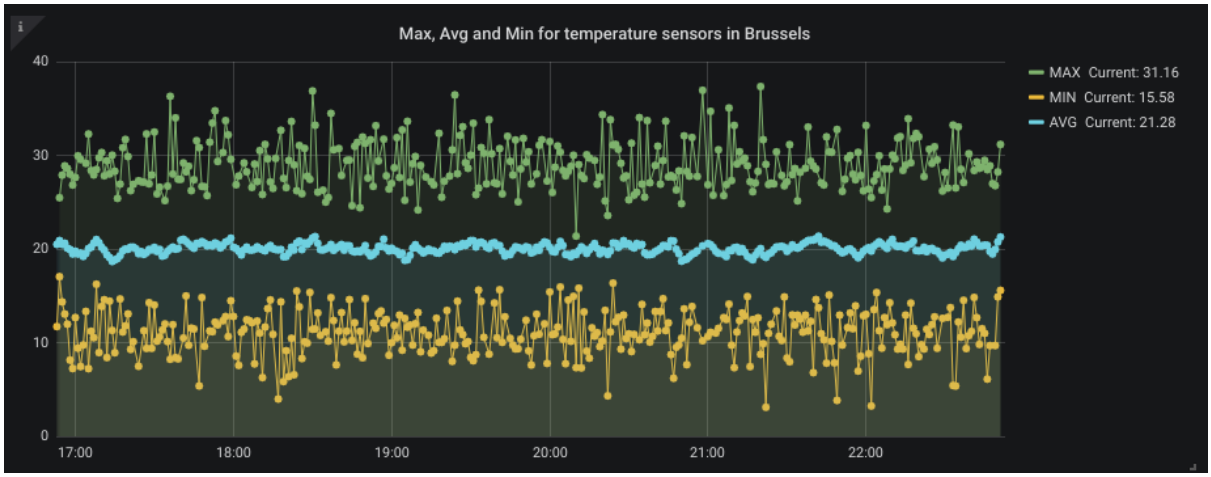

Figure 4.1: Max, Min and Avg temperature in Brussels for the last 6h, rolled-up by every 1m

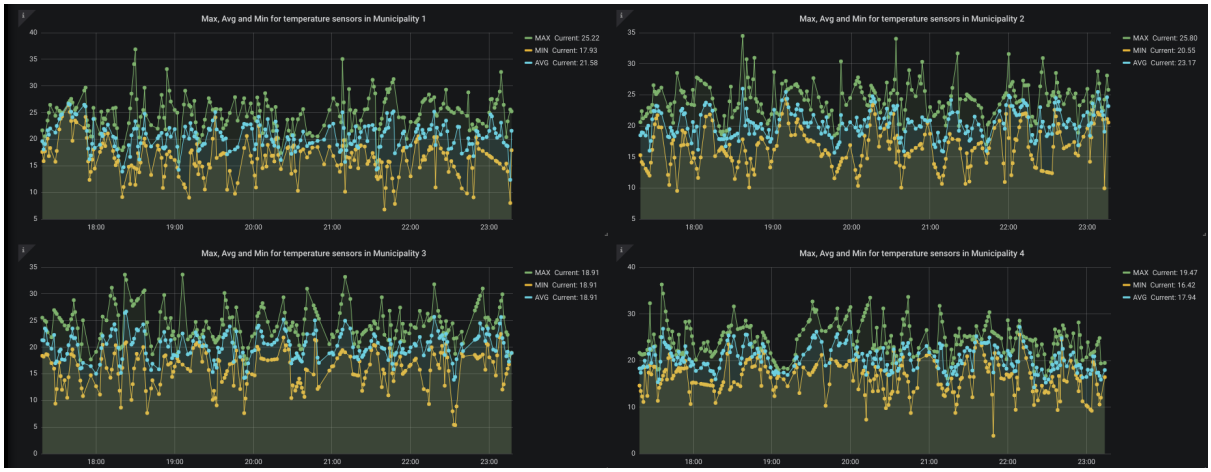

Figure 4.2: Max, Min and Avg temperature in 4 different municipalities for the last 6h, rolled-up by every 1m

Light sensors

In this graph we also show the results of multiple queries (min, max and average), but this time the data represents the measurements of light sensors. The rolled up interval is also 1m and the range of time shown is 6 hours. As well as temperature data we show two different level of space granularity: Brussels and Municipality. One important observation we can make is the fact that the average is very close to 40, as expected from the data generator distribution.

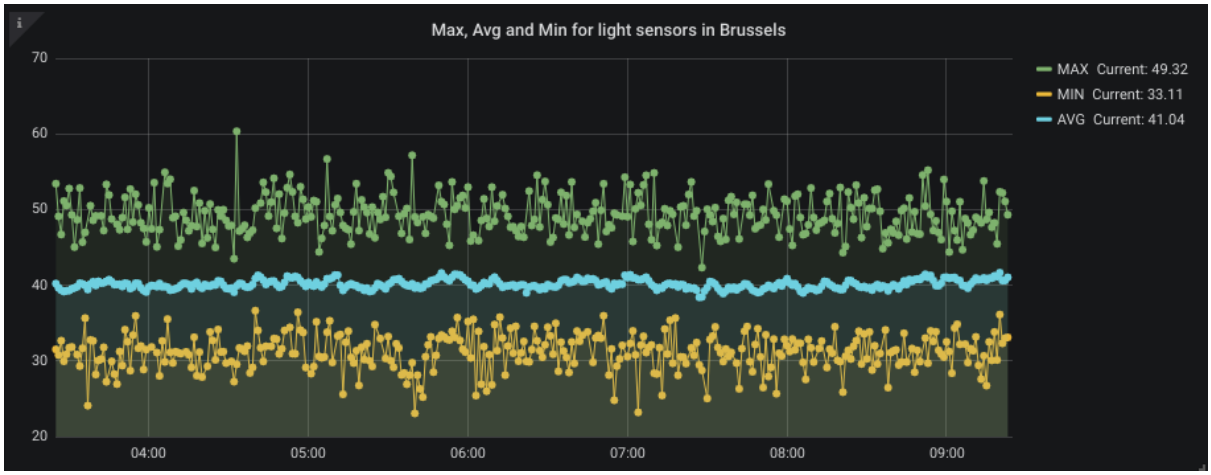

Figure 4.3: Max, Min and Avg light in Brussels for the last 6h, rolled-up by every 1m

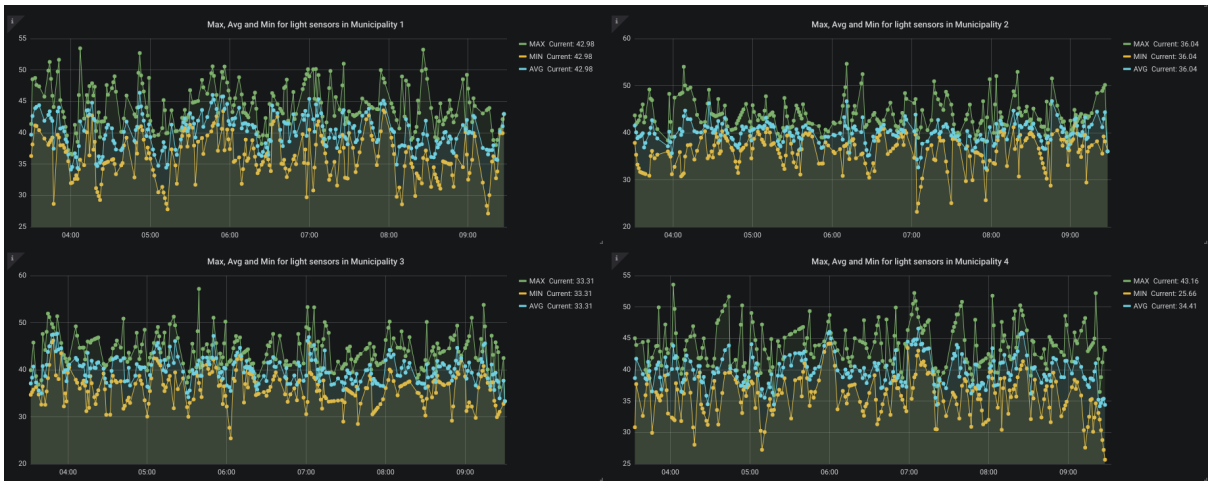

Figure 4.4: Max, Min and Avg light in 4 different municipalities for the last 6h, rolled-up by every 1m

Humidity sensors

The roll up intervals and the window of time that is shown in the following graphs is the same as for temperature and light. You will just notice the different values for average, min and max since the data is described by a different distribution.

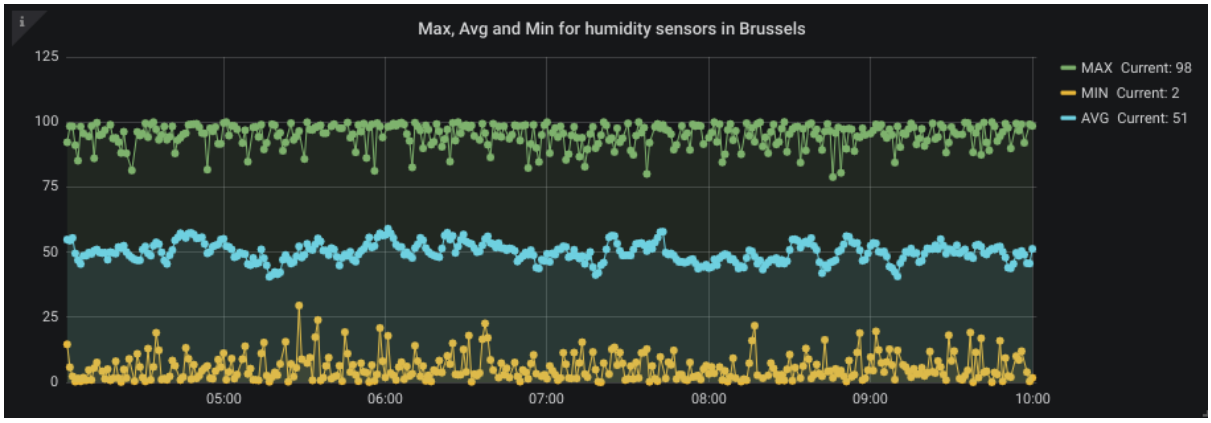

Figure 4.5: Max, Min and Avg humidity in Brussels for the last 6h, rolled-up by every 1m

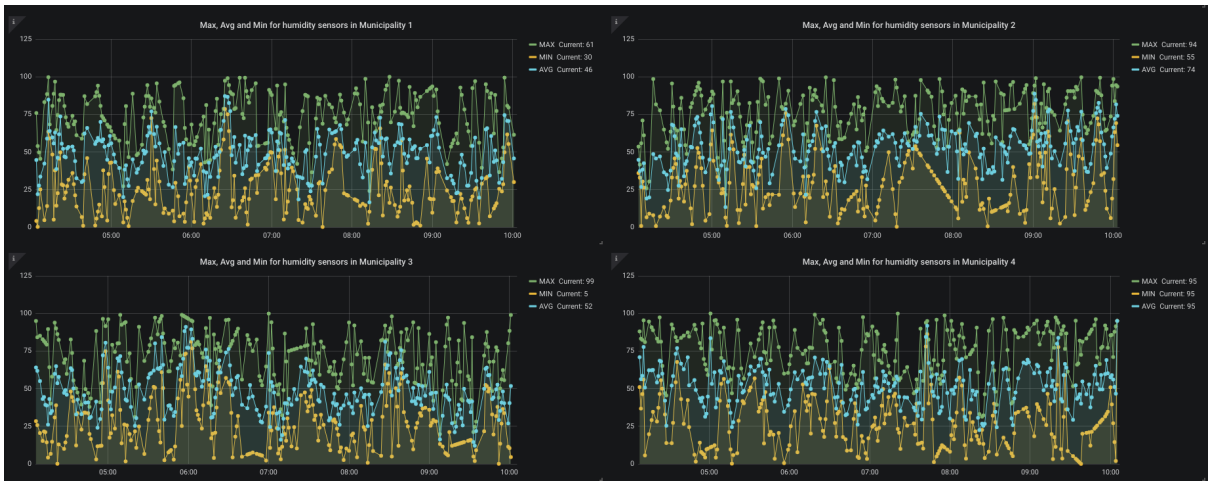

Figure 4.6: Max, Min and Avg humidity in 4 different municipalities for the last 6h, rolled-up by every 1m

Motion sensors

The figures below show the result of the measurements of motion sensors. For this type of sensors a range of the last 24 hours is shown. This sensors basically says if there is movement or not by sending a 0 or a 1 respectively. We show the min, max and average and we rolled up the data within an interval of 15 minutes. So each point in the average corresponds then, to the average amount of movement in Brussels (figure 4.7) or a municipality (figure 4.8). In this case is almost 50 percent since the distribution is binary.

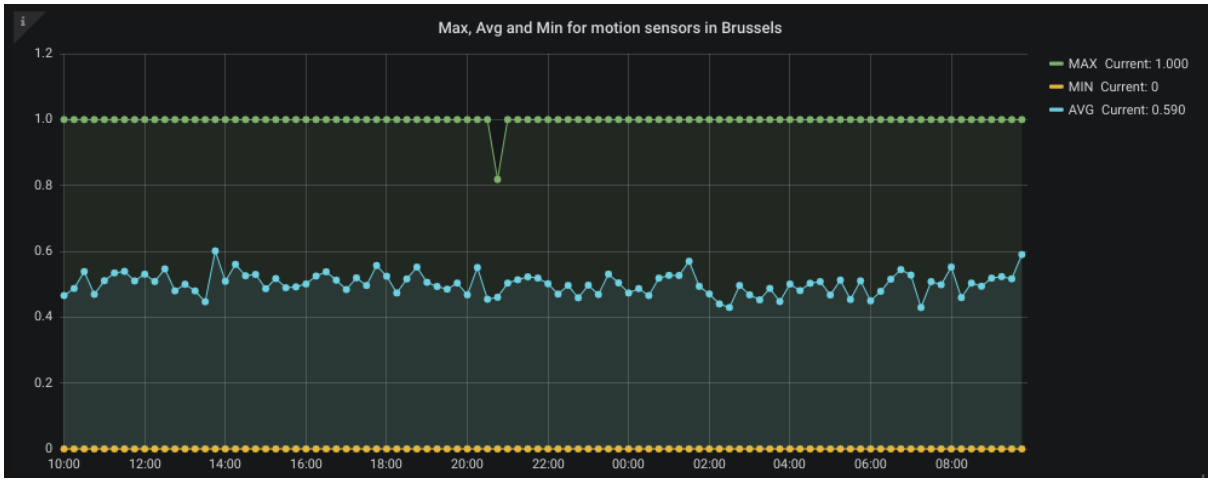

Figure 4.7: Max, Min and Avg motion in Brussels for the last 24h, rolled-up by every 15m

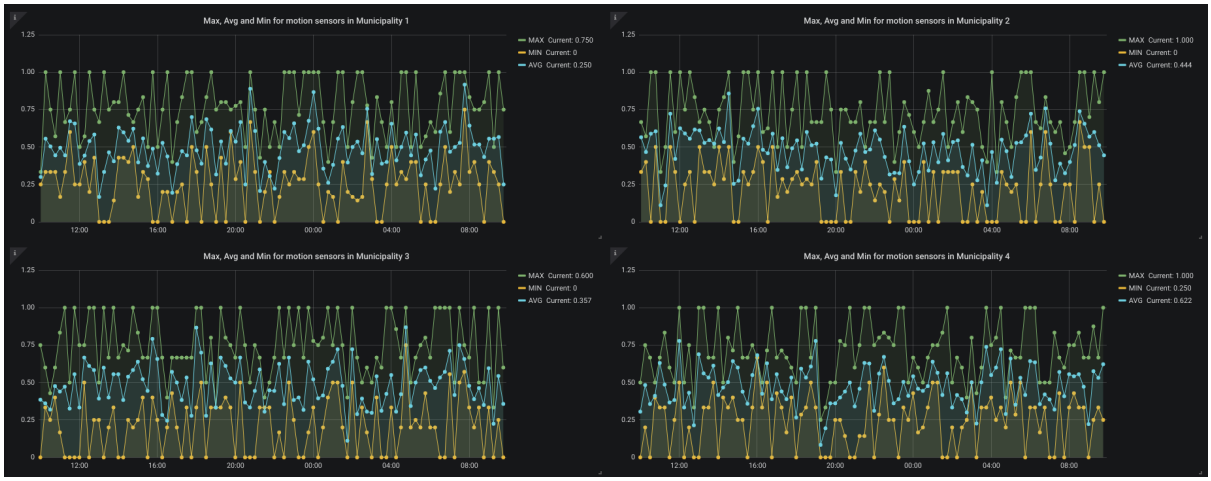

Figure 4.8: Max, Min and Avg motion in 4 different municipalities for the last 24h, rolled-up by every 15m

4.1.2 Daytime or Nighttime based on temperature

This is a very interesting query that we decided to show in a heat map graph. Basically we defined a threshold for temperature that is below and above 19.5 degrees and and we assigned colors to them blue (night-time) and red (day-time) respectively. Initialize we were asked to work with 15 minutes slots for 24 hours but since we let this open to work with different slot size, we decided to show slots of 1 minute and 1 hour because it is easy (for now, once the system has more data the result is the same) to visualize the result of the query. There is no a clear distinction between day and night regarding the hours because of the data distribution but in a real scenario this could clearly show a difference with respect to the hours, it also depends on the threshold.

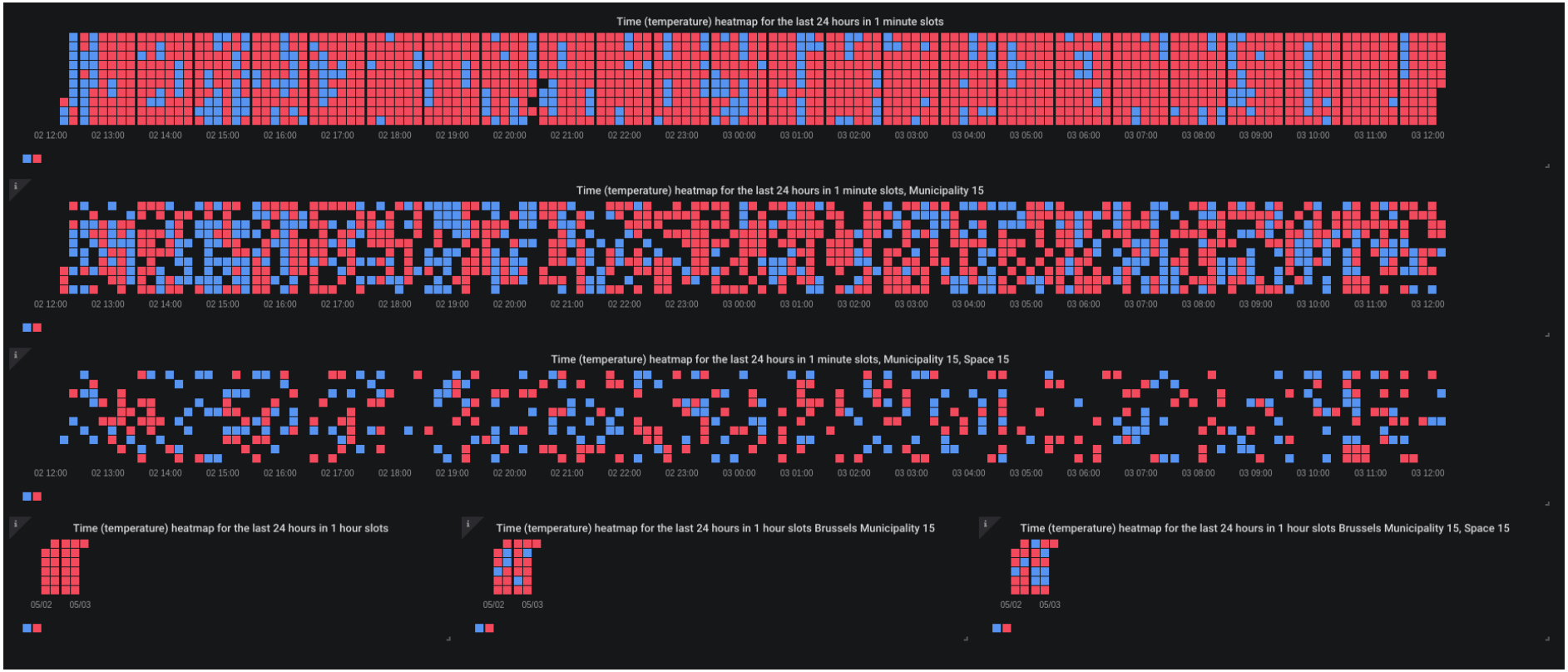

Figure 4.9: The red color represents day-time and blue night-time. In the first row we found a Time(temperature) in Brussels for the last 24h, in slots of 1m. In the second one we observe the same slots but the granularity change to a municipality level. In the third row again the same slots but now it shows a space granularity. In the last row we changed the slots size to 1 hour so you can see three different space granularity (previous graphs) in the last 24h.

4.1.3 Most frequent temperature analysis in a streaming fashion

This query is also a very interesting one because it is possible to observe the behaviour of the data in time, which in fact, matches with the properties of the data distribution we used to generate the data. In the second row of figure 4.10 we can see that all the yellow bars are approximately 20 which corresponds to the mean of the data distribution. And we aggregated the histogram of all the received data which in fact shows the behaviour of a Gaussian distribution.

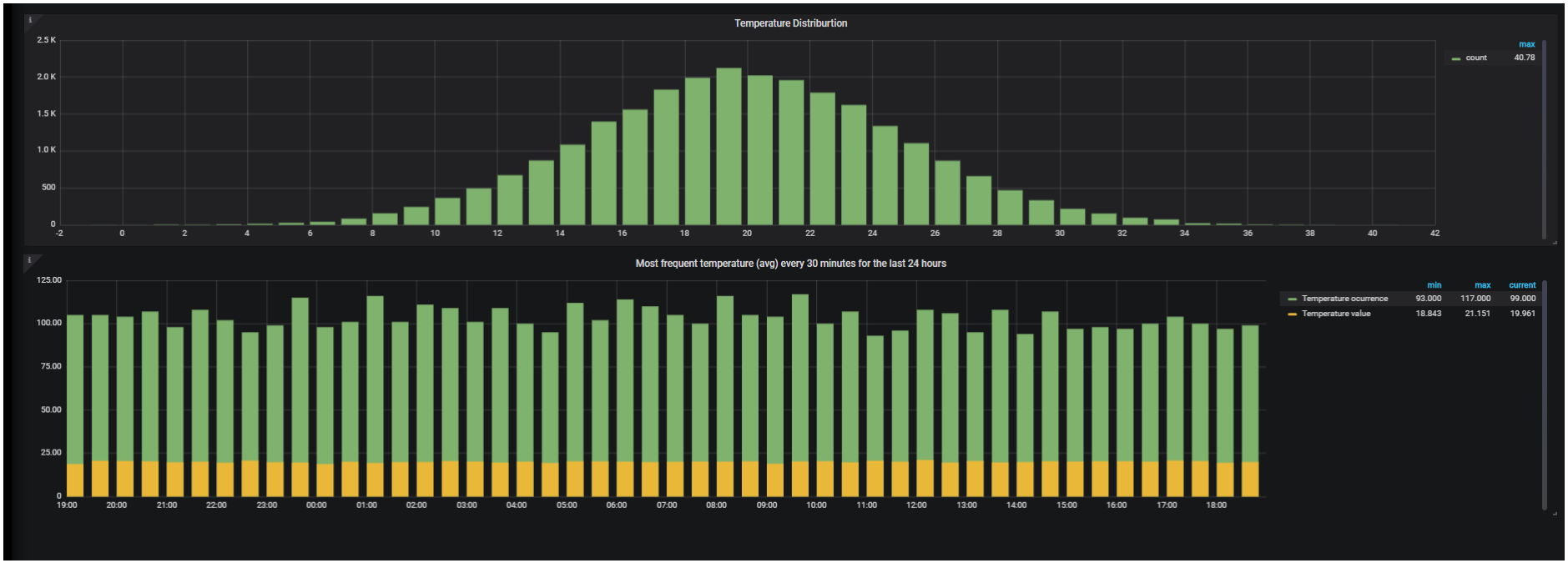

Figure 4.10: This graph shows in the first row the temperature distribution of the temperature sensors in Brussels. In the second row we can observe the average most-frequent temperature every 30 minutes. The yellow bars represent the common temperature and the green bars the frequency.

Chapter 5

Conclusions

The aim of this work was to design a scalable BDMA Pipeline for a smart city, which is capable of storing and treating data generated by a big number of sensors in a distributed fashion. To fulfill this purpose, the design strategy implemented for this work focuses on three key features: (i) The BDMA Pipeline has to be ready to scale up, (ii) it has to be fault-tolerant and (iii) data must be queryable as fast as possible. These requirements led us to design a BDMA Pipeline based on the λ-architecture. One of the major difficulties to set-up this architecture was to establish the communication between the different building blocks of the pipeline. This happened sometimes for version compatibility issues and other times for miss configurations. To ease this task, it has been decided to set-up all the needed software in a Docker environment. This allowed us to treat each software as independent and scalable services. Once the BDMA pipeline set up, the focus of the work lied on finding an efficient way of pre-computing queries that are expected to be done. In order to guarantee query velocity, the pre-computed queries only aggregate data in space, but not in time to allow query flexibility for the user. We decided to compute the time rollup operations on user demand using a time-series database (i.e OpenTSDB) to efficiently perform time-related queries. Afterwards, a Grafana dashboard has been chosen for the user to directly interact with this database. This allows the user to design its own queries in an easy and interactive way. The results presented in this project are extracted from such a Dashboard. Finally, a volume analysis has been done in order to estimate the required storage space needed for an eventual user of this BDMA pipeline. This analysis shows that the required storage increases over time and with the number of sensors. This is why a formula has been derived to evaluate the storage needed for a given number of sensors distributed in the city of Brussels.

Bibliography

[1] N. Marz and J. Warren, *Big data: principles and best practices of scalable real-time data systems*. Shelter Island, NY: Manning, 2015, oCLC: ocn909039685.

[2] "Apache Kafka." [Online]. Available: https://kafka.apache.org/quickstart

[3] "Welcome to Apache Flume — Apache Flume." [Online]. Available: https://flume.apache.org/

[4] "HDFS Architecture Guide." [Online]. Available: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

[5] "Spark Streaming | Apache Spark." [Online]. Available: https://spark.apache.org/streaming/

[6] "OpenTSDB - A Distributed, Scalable Monitoring System." [Online]. Available: http://opentsdb.net/

[7] "Grafana - The open platform for analytics and monitoring." [Online]. Available: https://grafana.com/

[8] "Apache flume - The open platform ingesting data." [Online]. Available: https://github.com/apache/flume/

[9] "Apache flume - Fetching data." [Online]. Available: https://www.tutorialspoint.com/

[10] "How to build a big data pipeline." [Online]. Available: https://dzone.com/