Migration of the business processes to distributed environment

IT4BI MSc Thesis

Student: SRDJAN NIKITOVIC
Advisor: ANTONIO CEBRIÁN (ENERBYTE)
Supervisor: ALBERTO ABELLÓ GAMAZO

Master on Information Technologies for Business Intelligence
Universitat Politècnica de Catalunya

Barcelona
July 31, 2016

A thesis presented by SRDJAN NIKITOVIC
in partial fulfillment of the requirements for the MSc degree on
Information Technologies for Business Intelligence

Abstract

In today world, with massive storage and computing power available at low cost, due to the expansion of cloud computing, many companies are starting their own businesses, processing large quantities of data and providing interesting services to third parties. Enerbyte is one of those companies, dealing with storing and processing energy consumption data coming from sensors.
Due to the nature of data, Enerbyte needs reliable, efficient and scalable data platform that can handle huge data quantities with lowest possible latency. This project aims to migrate all the current Enerbyte data processes to a new distributed data platform.

Contents

1 Introduction **1**

2 Current data platform and problem statement **2**
2.1 Data Sources . . . 3
2.1.1 Wibeee . . . 3
2.1.2 Wattio . . . 4
2.2 Problem statement . . . 4

3 Architecture definition **5**
3.1 Technologies demanded by Enerbyte . . . 5
3.1.1 Apache Cassandra . . . 5
3.1.2 Apache Spark . . . 7
3.1.3 Apache Kafka . . . 7
3.2 Lambda architecture . . . 8
3.3 Final architecture . . . 9

4 Implementation **10**
4.1 Wibeee implementation . . . 10
4.1.1 Wibeee raw data model . . . 10
4.1.2 Wibeee hourly data model . . . 11
4.1.3 Wibeee batch and speed layer . . . 12
4.1.4 Wibeee serving layer . . . 13
4.2 Wattio implementation . . . 15
4.2.1 Wattio raw data model . . . 15
4.2.2 Wattio hourly data model . . . 15
4.2.3 Wattio batch and speed layer . . . 15
4.2.4 Wattio serving layer . . . 16

5 Deployment of the given architecture **18**
5.1 DC/OS platform test . . . 19
5.2 Datastax platform test . . . 20

6 Experiments and discussion **22**
6.1 Environment configuration 22
6.2 Wibeee batch and speed layer tests 23
6.3 Wattio batch and speed layer tests 24
6.4 Recommendations for higher workload 25
6.5 Platform costs 25

7 Conclusion **26**

A Creating wibeee hourly data access object **27**

B Wibeee Spark batch process logic **29**

C Wattio Spark batch process logic **32**

References **34**

List of Figures

2.1 Wibeee device 3
2.2 Wattio device 4

3.1 Cassandra horizontal scalability 6
3.2 Lambda architecture 8
3.3 Lambda architecture in Enerbyte 9

4.1 BPMN Diagram: Wibeee pipeline 12
4.2 Problem that occurs in Wibeee speed layer 13
4.3 Wibeee hourly logic 14
4.4 BPMN Diagram: Wattio streaming logic 16

5.1 Datastax Enterpise platform architecture 21

6.1 Amount of data preserved in memory in *Wibeee* streaming process 24

A.1 BPMN Diagram: Creating Wibeee Hourly object 28

B.1 BPMN Diagram: Wibeee batch logic 29
B.2 Base wibeee data 30
B.3 Batch wibeee data 30
B.4 Base and Batch datasets joined 31

C.1 BPMN Diagram: Wattio batch logic 32

List of Tables

4.1 wibeee_raw data model 11
4.2 wibeee_hourly data model 11
4.3 wibeee_batch_snapshots data model 14
4.4 wattio_raw data model 15
4.5 wattio_hourly data model 15
4.6 wattio_batch_snapshots data model 17

6.1 Wibeee batch and speed layer tests 23
6.2 Wattio batch and speed layer tests 24

Chapter 1

Introduction

The company hosting the Master Thesis, Enerbyte [1], is a start up company which deals with processing, storing and analyzing of data related to energy market. Enerbyte operates in a B2B [2] and B2G [3] models, in which it offers its data platform to smart cities and energy utilities, which they further offer to final users in order to strengthen their market position. Enerbyte's main product is *Virtual Energy Advisor*, a mobile and web application which provides final users relevant insights into their energy consumption. The following information are provided to product users:

- how much electricity they spent within a specified time interval;
- how much money they spent within a specified time interval;
- how to optimize their electricity consumption and achieve money savings;
- whether changing tariff can help them to save money;
- what is their forecasted consumption until the end of a specified time interval;

The goal is to expand the application, introducing marketplace, neighbor comparisons, disaggregation algorithms, etc. *Virtual Energy Advisor*, besides providing interest insights to final customers about their energy consumption habits and routines, helps utilities and smart cities manage the electricity supply, reduce peaks and optimize energy grids and networks. All in all, it helps energy markets to become more efficient.

[1] www.enerbyte.com
[2] http://www.investopedia.com/terms/b/btob.asp
[3] http://www.investopedia.com/terms/b/business-to-government.asp

Chapter 2

Current data platform and problem statement

Although Enerbyte is a startup company, the first data platform they developed was intended to be used as a proof of concept, but was not intended for large production deployments with time series data for thousands of customers Enerbyte expects to have. We speak about time series data [1], because all data sources Enerbyte uses send measurements related to same variable in regular time intervals.
Currently Enerbyte stores data in MySQL relational database, that, in a single instance mode, is not meant to store huge quantities of time series data. Some of MySQL shortcomings are:

- Cannot handle thousands of write operations per second;
- Cannot handle thousands of read operations per second;
- Does not easily scale out;
- Does not support parallel data processing;
- Enforces *ACID* properties which entails a cost when such properties, as in Enerbyte usecase, are not needed;

Source data is ingested to MySQL using a *REST* API. Reports that *Virtual Energy Advisor* provides use aggregated data, at different levels of granularity. In order to compute aggregated consumption tables, schedulers execute SQL statements wrapped in PHP code, which perform ETL process. ETL process reads from source tables, performs aggregations and insert into hourly and daily electricity consumption tables.
Since ETL process is executed in specified time interval, it causes high latency between data generation and data visibility in web or mobile applications. Maintaining the process became extremely difficult, because data pipelines are part of the legacy system and no documentation for them is available. Also, further evolution and improvement of the system became almost impossible due to the fact that all the components of the system are *tightly coupled*, which is opposite to the good practices for service oriented architecture [2]. Most important shortcoming of the existing data processing platform is scalability; the impossibility for parallel data processing imposes a problem when amount of data increases.
In addition to data processing issues, there is another reason why Enerbyte had to change to a new data

platform: *business requirements*. Utility companies and smart cities are willing to cooperate and establish partnerships only with companies who have data infrastructure able to easily scale out as well as a *loosely coupled* system architecture based on *micro services*.
Changing the data infrastructure became a highest priority task for a data driven organization like Enerbyte.

2.1 Data Sources

Enerbyte has three main data sources:

- *Bills*: utility companies provide Enerbyte access to monthly electricity consumption data for each final user;
- *Smart meters*: also provided by utility companies, send hourly electricity consumption data to Enerbyte. Smart meters require special hardware infrastructure in user's home, that not all of the users have;
- *Devices*: alternative for homes that do not have infrastructure that supports smart meters installation. Enerbyte processes data from two device types: *Wibeee* [1] and *Wattio* [2];

Migrating data processing pipelines for *wibeee* and *wattio* was the main topic of the thesis.

2.1.1 Wibeee

Wibeee, shown on figure 2.1, is electricity consumption analyzer with a wi-fi connection, that sends minute

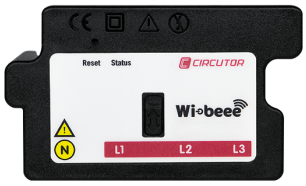

Figure 2.1: Wibeee device

consumption data. Values that it sends are *cumulative*: when the device is installed, it starts from zero, and every following minute it sends a value that is larger than the previous data point for consumption that user has achieved within a minute interval. There are multiple issues that occur:

- users switch off wi-fi for an unpredictable time period, which causes data delays. After wi-fi is switched on, there is a large bulk of data that arrives that needs to be properly handled;
- due to unpredictable behavior of http connection, data can arrive unordered;
- users sometimes accidentally reset the device, which sets the counter to an unpredictable value, and subsequent consumption values arrive starting from the resetted counter value;

[1] http://wibeee.circutor.com/
[2] https://wattio.com

2.1.2 Wattio

Wattio, shown on figure 2.2, is also installed in final users home and it also uses wi-fi to send consumption

Figure 2.2: Wattio device

data. The format in which it sends data as well as time interval differ: it sends data every 15 minutes; values that it sends stand for expected consumption for a next 1 hour interval. Each device reading is isolated from previous and does not affect future data readings, so there is no issue when device is resetted which is the case with *wibeee*. But, data can still arrive delayed and unordered.

2.2 Problem statement

The new data platform that Enerbyte wants to develop needs to satisfy the following non functional requirements:

- *Capacity*: it needs to be able to store up to petabytes of data;
- *Scalability* and *Maintanability*: the system should scale out with minimum human interaction;
- *Performance*: read and write operations should be executed with minimum possible latency. It should be able to process around 50000 *wibeee* and 50000 *wattio* device readings with smallest possible latency. This is the requirement introduced by Enerbyte's management, since that is the amount of users Enerbyte will have in foreseeable future;
- *Recoverability* and *Reliability*: In case of hardware or software malfunctions, platform needs to be able to continue functioning normally without affecting user experience;
- *Availability* and *Data Integrity*: In case of failures, no data should be lost. Also, all data belonging to a single Enerbyte project (also referred as *tenant*), which stands for a utility company or a smart city Enerbyte cooperates with, needs to be either stored on separate hardware or needs to be *conceptually* isolated from data belonging to other projects in order to comply with legislation procedures;

Chapter 3

Architecture definition

Firstly, technologies that Enerbyte wanted to use will be talked about and why they fit well to the Enerbyte use case will be analyzed. After that, we will speak about *lambda architecture* and why we decided to implement it in the company. In the end, final data architecture in Enerbyte will be presented.

3.1 Technologies demanded by Enerbyte

From the beginning, Enerbyte expressed desire to use Apache Cassandra [1] for storing time series data, Apache Spark [2] as a data processing framework and *Apache Kafka* [3] as a distributed messaging system. In the following sections, we will elaborate why all those technologies are excellent choice for Enerbyte's use case.

3.1.1 Apache Cassandra

Apache Cassandra, built on Amazon Dynamo [4] and Google Big Table [5], is a distributed column-family database for storing large amounts of data across commodity servers, while providing high availability and no single point of failure. In the CAP theorem [6], it stands on *AP axis*, which means it offers high availability and partitioning, giving up consistency and introducing the term *eventual consistency*, which is the feature that Enerbyte, based on use case, can adopt.
It has scalability like no other NoSql system, as presented on figure 3.1 This means that Enerbyte can easily add more nodes to the cluster when storage demands increase, without affecting overall performance of the system. After new node is added, each of the existing Cassandra nodes redistribute some of the data they store so that cluster remains balanced.
Cassandra uses hash function to distribute data across the cluster. Each node has a dedicated token range,

[1] http://cassandra.apache.org/
[2] http://spark.apache.org/
[3] http://kafka.apache.org/
[4] https://aws.amazon.com/documentation/dynamodb/
[5] https://cloud.google.com/bigtable/
[6] http://robertgreiner.com/2014/06/cap-theorem-explained/

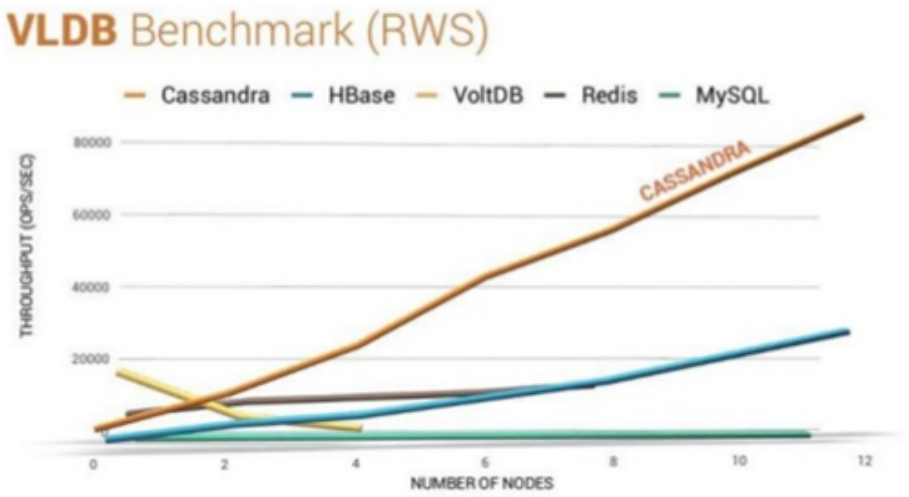

Figure 3.1: Cassandra horizontal scalability

which represents the collection of hash values that it is responsible for storing. It has a primary key composed of two parts:

- *Partition key:* one or multiple fields from our table schema on which hash function is applied, based on which a tuple is distributed in the cluster;
- *Clustering key:* one or more fields of our schema based on which Cassandra orders tuples within a partition;

Due to these features, it is clear that it has fine grained support for storing and processing time series data after putting *wibeee* and *wattio* device identifiers as partition keys, so that all data belonging to a single device is stored together, and time column as a clustering key so that all the values are ordered by time. Accessing specific time interval only requires getting the partition and finding the start date. After, only sequential scan is performed. This way, most expensive part of the *read* operation is avoided - *random access*.
Failover is prevented by introducing *replication*, which functions without any necessity for user interaction. User only needs to specify desired replication factor, for production purposes recommended factor is two or three, and in case of node failures our application is still available and our data is not corrupted. When the node is again up and running, Cassandra automatically redistributes corresponding token ranges back to the node that was *down*. It also offers multi-datacenter replication, where our data is consistent even in case of regional outages.
Cassandra has *peer to peer* architecture, that makes it extremely fast at writing. Query can be submitted to any node, and that node will forward the query to a node that has data, if it is a read operation, or to a node that should store data that is being inserted, if it is a write operation.
Non functional requirement that data belonging to different projects needs to be conceptually or physically stored separately, can in Cassandra be achieved using separate *keyspaces*, which represent separate logical

schemas within the same Cassandra cluster. This way, *multitenancy* [7] can be enabled using shared hardware and software infrastructure to satisfy needs of different projects. In this paper, terms *project* and *tenant* are used interchangeably: Enerbyte has multiple *projects*, which are also refereed as *tenants* in our system, and each one of them will be stored in a separate Cassandra keyspace.
Cassandra also integrates well with Apache Spark, and in following section we will analyze why it is excellent data processing framework for Enerbyte's use case.

3.1.2 Apache Spark

In addition to Cassandra as storage system, Apache Spark will be used for data processing and machine learning that will be performed in the future. Spark is extremely fast engine for large scale data processing. It supports both batch and stream processing, so companies can either process their data in real time or at specified intervals like in conventional data warehouse systems. It is 10x faster in batch and 100x in memory comparing to Hadoop MapReduce paradigm [8], and it is top level project in Apache community. Spark has much richer set of data transformations to use as well as simpler programming model than MapReduce. When finely tuned, Spark can exploit data locality, so that Spark instance on a node processes only the data which is stored on that node, which makes entire process extremely fast.
Great benefit Spark offers is easy integration with Cassandra, by using product called *Spark Cassandra Connector* [3] developed by *Datastax*. This product makes Spark more appealing framework to use in Cassandra deployments comparing to others like *Apache Flink* [9]. Apache Flink, at the time of technology evaluation, did not have Cassandra connector so users would have to focus more on setting the connection and maintaining it's performance instead of focusing on their data pipeline logic.
Goal of Enerbyte is to have as low latency as possible between data generation from above mentioned devices, until data being available in aggregated tables from which reports for final customers are being generated. Here is where Spark proves to be very effective, because it has Streaming component, which enables data that arrives from the devices to be processed in real time.

3.1.3 Apache Kafka

Spark Streaming needs efficient ingestion system which will serve as a buffer for source data from devices, from which Streaming will consume data at a pace it can process. Among different alternatives Enerbyte decide to use Apache Kafka. It stores data in *topics*, and it:

- Easily integrates with Spark;
- Replicates data over multiple nodes, so it is resilient to node failure;
- Can handle huge write load, although it can be considered as *distributed, replicated and partitioned commit log*;
- Supports partitioning the topic which enables parallel data processing;

In case of Spark Streaming process failures, it is easy to, by implementing direct Spark-Kafka integration [4], recover the process and continue reading from the topic where previous Spark process has crashed

[7] https://en.wikipedia.org/wiki/Multitenancy
[8] http://hadoop.apache.org/
[9] https://flink.apache.org/

before. It is also appealing to Enerbyte engineering team to use Kafka, although they could reuse it as a messaging system for other applications they plan to work on in the future.
Enerbyte management came to conclusion that using Spark Streaming data processing will also greatly strengthen Enerbytes market position, as all the reports users are looking will be generated with most up to date data, without latency that old batch data processing paradigm entailed. According to [5], efficient and reliable streaming process needs a batch supporting process, that can be executed at any time, and that can rebuild aggregated data used for reporting purposes from the source raw data. After analyzing best practices from above mentioned book, we realized that implementing λ architecture, is what Enerbyte needs.

3.2 Lambda architecture

λ architecture is best described on image 3.2. λ architecture introduces the following layers:

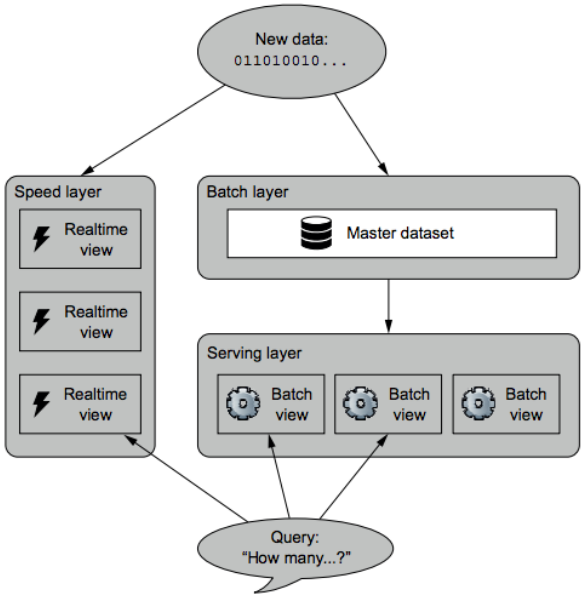

Figure 3.2: Lambda architecture

- *Speed layer*: real time data processing, that in Enerbyte will be implemented using Spark Streaming component;
- *Batch layer*: Ingestion process into *master data set* using Spark Streaming component. Master data set is Single source of truth in λ architecture. It contains raw data that was generated by *wibeee* and *wattio* devices;
- *Serving layer*: aggregated consumption data that is used in generating reports for the final user, and that can be reprocessed from *master data set* using Spark Batch component;

3.3 Final architecture

λ architecture that Enerbyte will implement, using Apache Kafka, Spark and Cassandra looks like on the figure 3.3 After data is ingested into messaging system, every 15 seconds Spark Streaming process starts

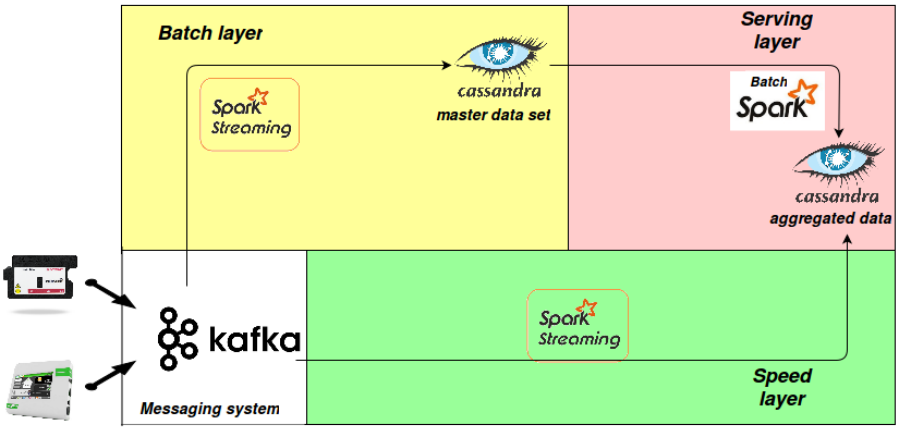

Figure 3.3: Lambda architecture in Enerbyte

for each of the *wibeee* and *wattio* devices. One branch of each streaming process inserts data into Cassandra raw tables, which represent *master data set* in λ architecture, while other branch performs real time aggregation and data quality resolution, and inserts data into Cassandra aggregated tables (*serving layer* in λ architecture). There is also a Spark Batch process developed to reprocess data from *master data set* into *serving layer*.

Chapter 4

Implementation

In this chapter, we will explain how *wibee* and *wattio* streaming and batch processes have been developed as well as Cassandra modeling for devices *master data set* and *serving layer*.

4.1 Wibeee implementation

Wibeee device, as mentioned in section 2.1.1, sends consumption data every minute to Enerbyte HTTP endpoint, and invokes a HTTP POST request which inserts data into Kafka topic called *wibeee*. We needed to choose the data type in which we are going to serialize the messages stored in Kafka. Among JSON, XML and Avro, we chose to use Avro due to recommended Kafka deployment practices [6]. In accordance to λ architecture paradigm, Spark Streaming process has been developed which consumes data from Kafka topic *wibeee*, processes it and injects into *wibeee raw* and *wibeee hourly* Cassandra tables.

4.1.1 Wibeee raw data model

Wibeee_raw is the Cassandra table where all the source raw data is ingested, without applying any transformations to it. It represents *wibeee* master data set in λ architecture paradigm, so that in case of streaming process failures or data corruptions, we can use that data to reprocess aggregate tables using Spark batch process.
When modeling table, several alternatives from Datastax best practices [7] for modeling time series data have been evaluated, and alternative number 1, *single device per raw* has been used. *Device id* represents partition key, and *datetime* represents clustering key as can be seen in table 4.1 Each of *ApparentPower*, *ActivePower*, *ReactivePower*, *FrequencyPhase*, *PowerFactor*, and *ReactiveEnergy* are *Cassandra user defined types* that contain 4 values, and correspond to different measures *wibeee* device is sending every minute. Enerbyte is currently interested only in is *ActiveEnergy* field, which contains electricity consumption information. Alternative number 2, *partitioning to limit raw size* was excluded although it is recommended in cases when we store data at a granularity smaller than a second (millisecond for example). Alternative number 3, *reverse order timeseries with expiring columns*, is suggested when we want to limit amount of data we want to store, for example only last month, since it introduces data expiration into table definition. Enerbyte needs to store all historical data for each user, so that is the reason why this alternative was

field name	field type
mac (partition key)	varchar
ip	varchar
soft	varchar
model	varchar
date_time (clustering key)	timestamp
apparent_power	ApparentPower
active_power	ActivePower
reactive_power	ReactivePower
frequency_phase	FrequencyPhase
power_factor	PowerFactor
active_energy	ActiveEnergy
reactive_energy	ReactiveEnergy

Table 4.1: wibeee_raw data model

field name	field type
mac (partition key)	varchar
date_hour (clustering key)	timestamp
minute	varchar
model	int
consumption	float

Table 4.2: wibeee_hourly data model

excluded.

4.1.2 Wibeee hourly data model

Wibeee_hourly table represents the consumption made by each device within each hour. Hour granularity table is the most important table for reporting in Enerbyte data model, out of which all reports will be produced, so it represents *serving layer* in λ architecture paradigm. All the reports that Enerbyte displays to the final user include time dimension (electricity spent within a day, month, year), and for all of them this table will be used. After evaluating Datastax best practices, same alternative has been used as in *wibeee_raw* table to model *wibeee_hourly*, as can be seen in table 4.2. In both *wibeee_raw* and *wibeee_hourly* tables, partition key is *mac*, so all the data belonging to the same mac address, which is unique identifier for a *wibeee* device, will be stored together. Clustering key specifies the order in which data belonging to the same partition key will be stored. In *wibeee_raw*, clustering key is a *date_time*, which is full timestamp, while in *wibeee_hourly*, clustering key is *date_hour*, which is hourly granularity of *date* data type. Both clustering keys are by default in ascending order.
Key benefit of these features that Cassandra is offering is that for a date range query for a certain mac, there will be a sequential scan of data after node where data is stored is determined applying hash function on partition key. This way, very expensive random reads are eliminated, and read performance is very high.

4.1.3 Wibeee batch and speed layer

Enerbyte has multiple projects, in different parts of Europe, so non functional requirement, as explained in section 2.2, is that data belonging to one project is separated from data that belongs to other projects. This is modeled in Cassandra by creating multiple *keyspaces*, as stated in section 3.1.1. Keyspace information is encoded in Kafka message, so once data pipeline reads data from Kafka, it has to split the results into different final destinations - *keyspaces*. BPMN Diagram on figure 4.1, conceptually represents data pipeline

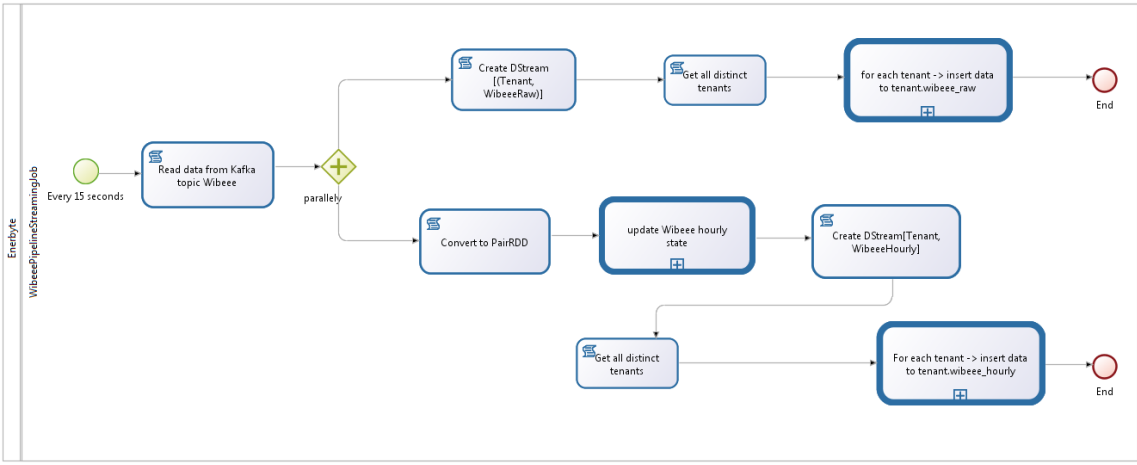

Figure 4.1: BPMN Diagram: Wibeee pipeline

design. After reading data from Kafka, parallely we perform processes for *wibeee_raw* and *wibeee_hourly* tables. In *wibeee_raw* path, for each of the elements in Spark Streaming micro batch, we create data access objects that match Cassandra table *wibeee_raw* definition in table 4.1 and insert data into Cassandra.
In *wibeee_hourly* path, we also need to create data access object for each of the elements so that they match *wibeee_hourly* table. As mentioned in chapter 2.1.1, *wibeee* device sends cumulative values of user energy consumption. In order to be able to compute what was the consumption for each device in a specific hour, we need to subtract cumulative consumption value in the last minute of that hour, with the consumption of the last minute in hour before. In order to keep the process fast, the best approach would be to keep additional data structures in memory, in order to be able to efficiently compute consumption information. We decided to use Spark's statefull transformation called *updateStateByKey()*, which preserves data structure called *state* in memory, so we can decide which data we want to have available in every new micro batch. Entire process of creating *wibeeeHourly* data access object is available in Appendix A. This way, in real time we process energy consumption of each device.
The data quality issue that streaming pipeline does not resolve are *gaps* (missing data) in *wibeee* readings. For example, if data does not arrive for two hours (due to problems in Internet connection, device itself, etc) as shown on figure 4.2, all the generated consumption for those hours will be accumulated in the first new data point that arrives. Our streaming process would put all the accumulated consumption in the hour new data point belongs to, while previous *missing* hours would have zero consumption. So, in our example, consumption of 120 units (142 units in new data point minus 22 units in previous data point) would be inserted to hour 16 instead of being distributed to hours 14, 15 and 16. This error caused us to develop

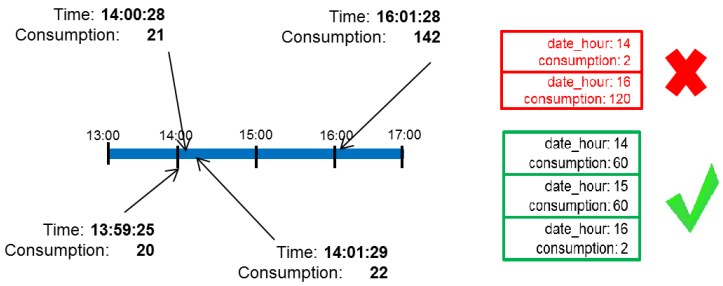

Figure 4.2: Problem that occurs in Wibeee speed layer

wibeee Spark batch process, whose purpose in λ architecture that Enerbyte implements is to fix gaps that occur in speed layer.

4.1.4 Wibeee serving layer

As stated in section above, the problem that *wibeee* speed layer cannot successfully solve are gaps in data arrival. In order to solve those problems, as well as be able to rebuild serving layer from the master dataset at any point, Spark batch process will be executed which performs *linear interpolation* [8] to resolve gaps (missing data points). When there is a *gap* in data arrival longer than an hour, process calculates consumption per minute by dividing consumption in a gap with number of minutes gap corresponds to, and then to each hour in a gap assigns appropriate value. This way, problem that is described on figure 4.2 is resolved.
Input parameters for batch process are the following:

- *Batch process start dateHour*: starting date time, at the granularity of hour, for which we want to execute batch process;
- *Batch process end dateHour*: end date time, at the granularity of hour, for which we want to execute batch process;
- *Keyspaces*: list of keyspaces for which we want to executed batch process
- *Macs*: List of devices for which we want to execute batch process. If no devices are specified, process will execute for all of the devices we have in the system for specified start and end hours;

If there were was no missing data in *wibeee master data set*, logic to implement batch process would look like on figure 4.3 For the wibeee batch process that has specified start and end hour, in order to calculate what is the consumption for the first hour, we need to subtract data consumption point in minute 59 of first hour (point B on figure) with consumption point in minute 59 in the hour before (point A in figure). For calculating the consumption in second hour, we need to subtract consumption point in minute 59 of second hour (point C in figure) with consumption point in minute 59 of first hour (point B in figure), etc.
But following two problems occur:

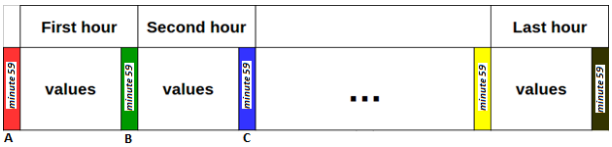

Figure 4.3: Wibeee hourly logic

field name	field type
mac (partition key)	varchar
start_batch_date_hour	timestamp
end_batch_date_hour (clustering key)	varchar
cumulative_consumption	float

Table 4.3: wibeee_batch_snapshots data model

- We do not know when exactly last data arrival, for every device, before batch start hour occurred. It could be exactly an hour before start hour of batch process, or it can be any other *timestamp* in the past;
- Readings of *wibeee* device occur every minute, so we consider certain hour for a device to have a proper reading if it has a data point in minute 59. If last reading in an hour happened in minute before 59, that means that there are gaps, which need to be corrected with batch process execution;

The first problem would cause us not to be able to determine what is the amount of data we need to process in order to perform batch process for specified start and end hour. We would need to iteratively scan for each device, until we find what is the hour prior to batch process start for which that specific device has readings. That would impose a high burden on our process and would result in huge performance impact.
Second problem would also impose problems in the process execution, because due to gaps, we do not always have data reading in minute 59 of each hour.
In order to resolve these issues, we introduced additional table in Cassandra schema called *wibeee batch snapshots*. This table is intended to keep the history of previously performed batch processes, and help speed up execution of any new scheduled batch process. It's data model is presented in table 4.3. Whenever batch process is executed for all the devices, we write to this table start and end hour of batch process, as well as cumulative consumption of each device that corresponds to last data arrival within that batch interval. This table serves as a snapshot of batch processes history, so that data engineers by executing simple queries can get insights in all previously performed batch processes. It removes additional process documentation that data team would need to maintain. By obtaining *max(end_batch_date_hour)* value, they can find out what is the last correct state of *wibeee* serving layer. Also, when data engineers want to perform new batch process, they only need to perform it starting from *max(end_batch_date_hour)* value. Also, *cumulative_consumption* field corresponding to maximum *end_batch_date_hour* value can be used to resolve problem of not being able to determine how much data needs to be processed to execute batch for specified time interval. From now on, every batch process will be performed by aggregating data from both *wibeee_raw* and *wibeee_batch_snapshots* tables.
Logic of Wibeee batch process is available in Appendix B.

field name	field type
wattio_user_id (partition key)	varchar
date_time (clustering key)	timestamp
channel1	float
channel2	float
channel3	float

Table 4.4: wattio_raw data model

field name	field type
wattio_user_id (partition key)	varchar
date_hour (clustering key)	timestamp
minute0	float
minute15	float
minute30	float
minute45	float

Table 4.5: wattio_hourly data model

4.2 Wattio implementation

Wattio device, as mentioned in chapter 2.1.2, sends consumption data every 15 minutes to Enerbyte HTTP endpoint, and invokes a POST request which inserts data into a Kafka topic called *wattio*. In accordance with λ architecture paradigm, Spark Streaming process has been developed which consumes data from Kafka topic *wattio*, processes it and injects data into *wattio_raw* and *wattio_hourly* Cassandra tables.

4.2.1 Wattio raw data model

Wattio_raw Cassandra table, represents master data set for *wattio* device, and is presented in table 4.4. Energy consumption information is stored in *channel1*, *channel2* and *channel3* columns.

4.2.2 Wattio hourly data model

Wattio_hourly table, same as *wibeee_hourly*, stores consumption made by each device within each hour. Due to the fact that *wattio* sends data every 15 minutes as well as due to nature of the data, we have decided to use a slightly different modeling approach as presented on table 4.5. *Wattio_hourly* has four columns in which consumption is stored: *minute0*, *minute15*, *minute30* and *minute45*. Each column is going to store consumption data that has arrived within corresponding 15 minute interval. When a query is issued over this table, in order to obtain the consumption that has occurred in a specific hour for a specific user, values of the 4 columns need to be aggregated.

4.2.3 Wattio batch and speed layer

Similarly to *wibeee*, same non functional requirement that data belonging to different projects need to be stored in different keyspaces is required. BPMN diagram presented on figure 4.4 conceptually represents

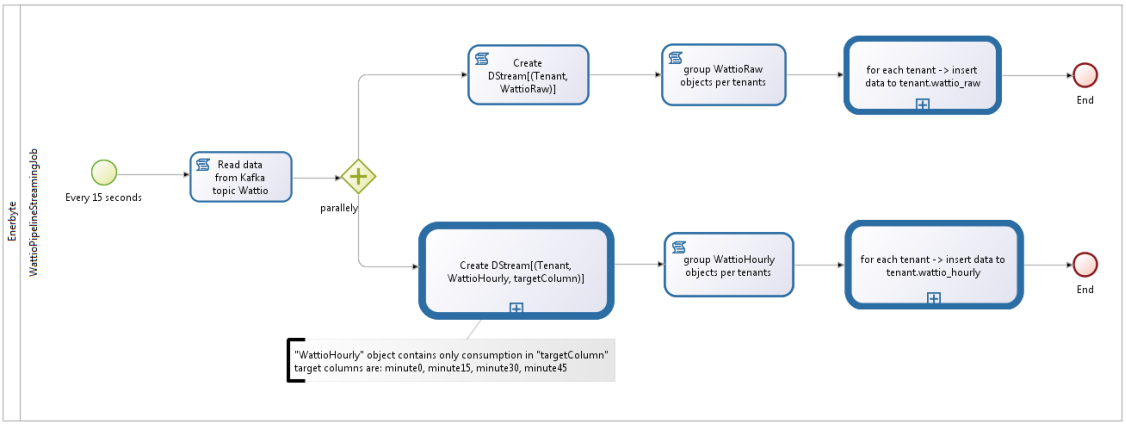

Figure 4.4: BPMN Diagram: Wattio streaming logic

data pipeline design. After reading data from Kafka, parallely we perform process for *wattio_raw* table:

- Creating data access object sequence of type *WattioRaw* which matched definition of *wattio_raw* Cassandra table;
- Group all the *WattioRaw* objects belonging to the same keyspace;
- For each keyspace, filters all the data that belong to that keyspace and inserting it into Cassandra table *wattio_raw*;

And for *wattio_hourly* table:

- Creating Data access object sequence of type *WattioHoury*. Whenever new Wattio measure arrives, we calculate the consumption and determine 15 minute interval to which that data element corresponds to. Based on the determined 15 minute interval, we return the target column (*minute0*, *minute15*, *minute30* or *minute45*) to the main flow;
- Group all the *WattioRaw* objects belonging to the same keyspace;
- For each tenant, from entire micro batch we filter data that belongs to each of the 15 minute intervals and we write it to Cassandra appropriate keyspace. We *cache* (temporarily preserve in memory) the RDD before processing, so these filter operations are very efficient;

4.2.4 Wattio serving layer

As opposing to *wibeee* device, there are no data quality issues that *wattio* streaming pipeline causes, as mentioned in section 2.1.2. But, as already stated, every streaming process needs to have a supporting batch process that can reprocess serving layer at any given moment. We decided to build batch process for *wattio* device as well.
Input parameters for *wattio* batch process are the same as for *wibeee* batch process: start and end hour, list

field name	field type
wattio_user_id (partition key)	varchar
start_batch_date_hour	timestamp
end_batch_date_hour (clustering key)	timestamp

Table 4.6: wattio_batch_snapshots data model

of devices and keyspaces.
As opposing to *wibeee* Spark batch process logic, where we needed additional Cassandra table called *wibeee_batch_snapshots* to be a part of process execution, *wattio* Spark batch process does not need such a data structure; *wattio* hourly consumption does not depend from previously received consumption values. But benefit of maintaining the state of our serving layer that data engineers in Enerbyte can use whenever they want to perform new batch process is why it has been decided to implement the same data structure as for *wibeee*, called *wattio_batch_snapshots*, whose model is presented in table 4.6 Batch process implementation is available in Appendix C.

Chapter 5

Deployment of the given architecture

Next step was to decide how to deploy the previously given solutions. Enerbyte does not have enough funds to invest into *in-house on premise solutions*, so a straightforward decision was to use a cloud service provider, specifically *Amazon Web Services* [1], due to team experience with the platform. We evaluated following alternatives for deploying Apache Kafka, Spark and Cassandra on Amazon:

- Provisioning EC2 instances, and installing and maintaining entire cluster by ourselves - concept known as *Infrastructure as a Service*. Even though this is the solution that many companies implement and enables full control of the cluster, it is not particularly attractive for small organizations due to large amount of administration work necessary to maintain it. Company would have to invest large amount of time and money in training in order to be able to maintain a cluster. It is the main reason why this solution was discarded since the beginning;

- Using *Instaclustr* [2] managed solutions, in the form of *Platform as a Service*, where within minutes we could provision Spark and Cassandra cluster, completely configured and ready to work. *Instaclustr* offers deploying their distribution on Amazon, which fits the needs of Enerbyte. It does not provide Kafka, so it would have to be managed and deployed by Enerbyte itself. Shortcoming of this solution is the license fee which is around 270 euros per month per node, without the price of Amazon instances that should host the solution. *Instaclustr*, even though it looked quite promising, was not affordable to Enerbyte;

- *Mesosphere Datacenter Operating System - DC/OS* [3], open source distributed operating system which uses *Apache Mesos* as its kernel. It offers a powerful Command Line and Graphical User Interface, to monitor, build and scale the cluster. Another strong point of the platform is that if offers all of the components that Enerbyte needs, Apache Kafka, Spark and Cassandra, as a *Platform as a Service*; they can be deployed and started to be used within minutes. In addition to them, it offers Apache Mesos as a cluster manager, which is able to more efficiently distribute resources in the cluster, as well as Apache Marathon, which enables efficient orchestration and strengthens running long time applications on top of Apache Mesos [9]. Due to all of points mentioned as well as the fact that there

[1] https://aws.amazon.com/
[2] https://www.instaclustr.com
[3] https://dcos.io

are no licensing fees as well as that deploying on Amazon is a straightforward process, we decided to test our data pipelines on *DC/OS*.

- *Datastax Enterprise* - Certified and licensed distribution of Apache Cassandra as *Platform as a Service*. In addition to Apache Cassandra, Datastax Enterprise offers deploying Apache Spark and Apache Solr instances in the cluster, as a part of Datastax Analytics and Datastax Search modules that come together with enterprise distribution. Deploying on Amazon Web Services is a straightforward process. It does not have Apache Kafka, which means that Apache Kafka would have to be deployed and managed by Enerbyte team. It provides a powerful graphical user interface for monitoring the cluster called OpsCenter. Also, particularly interesting to Enerbyte is Datastax Startup program [4], which offers Datastax Enterprise platform for free for qualifying startups that fulfill program conditions, which is the case with Enerbyte. As a part of startup program, Enerbyte would be able to get personalized support from Datastax experts for free. This point seemed especially appealing to the team, so we decided to test our data pipelines on Datastax Enterprise as well.

5.1 DC/OS platform test

Installing entire stack of technologies on Amazon was a quite straightforward process, and within minutes we were having a cluster up and running. First problems that we faced was related to architecture of DC/OS, where it introduced the concept of public and private agents [10]. Private agents are actually the nodes that would be running Spark, Cassandra and Kafka services, and they are accessible only from the public agents or from the administration zone. This means that all the additional services that Enerbyte needs to run in order to feed the applications with data stored on private agents would need to be deployed on public nodes, following DC/OS security and administration protocols. This was not attractive to Enerbyte team, although they already have established procedure and *end to end* deployment automation on regular Amazon instances.
Second issue that we faced was related to the number and type of machines necessary for cluster to run. In order to have a cluster of 3 machines running Spark and Cassandra workload, Enerbyte would need at least 6 running instances:

- *1 DC/OS master node* - which aggregates all the resources from all agent nodes and provides them to registered frameworks. For high availability in production environment it is recommended to run DC/OS with 3 masters instead of 1 [11];
- *3 DC/OS private nodes* - the nodes which will be storing data in Cassandra as well as processing it via Spark;
- *1 boot node* - relatively small AWS instance used to start up the cluster;
- 1 or more instances for DC/OS public agents, where other Enerbyte services would run;

Also, the recommended Amazon instance type to run Cassandra workload is m3.xlarge [5] for both agents and master nodes, which significantly affects the cost of the cluster.
After setting the cluster according to DC/OS recommendations, we started to test the pipeline performance

[4] http://www.datastax.com/datastax-enterprise-for-startups
[5] https://aws.amazon.com/ec2/instance-types

by forwarding Enerbyte simulators to the newly established cluster. In order for the streaming process to run successfully, and to be able to recover from Spark driver or executor failures, checkpointing needed to be enabled. Spark checkpointing is the process where Spark saves metadata information regarding the process to reliable storage. Also, checkpointing is required when statefull transformations, like *updateStateByKey* in our case, are used. Considering that the data platform was deployed on Amazon, using *Amazon Simple Storage Service* [6] was the logical solution, since storing Spark checkpoint information in Cassandra file system is still not available.
After configuring all the components and starting Spark streaming data pipelines, they ran correctly, but when streaming process was intentionally stopped, in order to stress-test recovery from S3 checkpointing, process was failing and was not able to restart correctly. We filled a bug report to DC/OS bug report portal [12], but no response or update has been received so far. Due to high number of necessary machines as well as high cost of them, administration complexity and platform unpredictable behavior, we have decided to proceed with Datastax Enterprise platform test.

5.2 Datastax platform test

Datastax offers production ready distribution of Apache Cassandra tested on hundreds of nodes, which is really important because Enerbyte will store *master data set* in Cassandra. It has less complex architecture than DC/OS (no Apache Mesos nor Marathon), so fewer Amazon instances is necessary. In order to have a three node cluster up and running, it was enough only to provision 3 Amazon instances, since Datastax runs Spark master collocated on one of the worker nodes.
When a Spark job is submitted in *cluster mode* [13], driver application runs on one of the nodes in the cluster and occupies at least 1 CPU unit of the total processing capacity. Although we also need minimum 1 CPU to run Spark executors, we came to the conclusion that we need at least 2 CPU cores per data pipeline. Although Enerbyte currently has two Spark streaming data pipelines, as well as due to the fact that some of the computing resources need to be available for the batch process execution, we came to the conclusion to test the platform performance using 3 AWS *c3.xlarge* [7] instances.
Although Datastax Enterprise does not provide Kafka as a service, we evaluated installing Kafka on Datastax Enterprise cluster of machines or creating separate Kafka cluster, on separate Amazon machines. After analysis of impact, pricing, future Enerbyte needs of the platform as well as recommended practices, we came to the conclusion to deploy Kafka as a separate cluster. We came to that conclusion due to following factors:

- Datastax enterprise comes as a well packed and tested solution, whose performance and compactness would probably be negatively affected by installing additional software components on the same machines;
- Enerbyte will use Kafka in the future as a distributed messaging service for their other applications, so it would be much more convenient to have it as a separate module in Enerbyte new data platform architecture, in order to reduce the burden of data platform for workload that is not meant to be processed or stored by it;
- For the workload that Enerbyte currently has, setting Kafka cluster of three *t2.small* machines should be more than enough, as tests show later, to satisfy current and foreseeable future needs. CPU and

[6] Amazon Simple Storage Service
[7] https://aws.amazon.com/ec2/instance-types/

memory per unit of *t2.small* instances are much cheaper than in *c3.xlarge* or *m3.xlarge* instances, so it would not impose high financial burden to the company;

Since entire solution is deployed on Amazon, S3 was again used as reliable storage for Spark streaming checkpointing.
After setting Kafka and Datastax Enterprise cluster each containing three machines, initial tests of the platform were performed, and all the processes (including critical and failing component in DC/OS: Spark streaming recovery from failure) were running successfully. We were intentionally restarting Spark and Kafka nodes, and both data pipelines were running without any errors, since Spark executor and driver processes were moved from nodes that were restarted to the nodes that are up and running. Cassandra is configured with replication factor 2, so in case when 1 node is restarted we have 1 node with exact copy of the data that can serve any given request to that data until restarted node is up again and until it gets all the corresponding data it was storing before it was restarted.
Due to all stated, Enerbyte decided to proceed with deploying Datastax platform. The final architecture looks like on the figure 5.1. Although *c3.xlarge* machines were used, each having 4 CPU cores and 7.5 GB

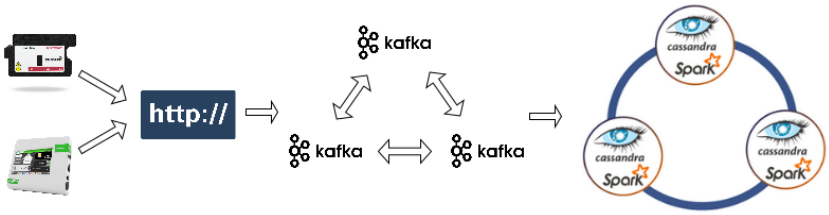

Figure 5.1: Datastax Enterpise platform architecture

of RAM memory, total number of CPU cores in our cluster is 12 with around 21 GB of memory. Datastax reserves one core and half of available memory per node for background operations, so total number of cores is 9 with total available memory of 12GB available for Spark workload.

Chapter 6

Experiments and discussion

Firstly, we will discuss the specific environment settings for the above mentioned stack of technologies. After, results of running streaming pipelines and scalability of the solution will be presented.

6.1 Environment configuration

In the solutions that consist of Kafka, Spark and Cassandra, in order to achieve high performance, it is necessary to finely tune each component while looking at them as a *whole integrated system*, not as *isolated instances*.
As mentioned section 3.1.3, Kafka supports dividing topic into multiple partitions, and always tries to store them on different nodes, called *brokers*, of the cluster. Choosing number of partitions for a topic is a complex problem [14], but in order to exploit parallelism that Kafka partitions offer, it is also necessary to have appropriate number of consumers to read from the partitions. When integrating Spark with Kafka [4], our Spark application represents the consumer group, which has as many consumers as there are executors given to the application at launch time [15]. Also, each Spark executor needs to have at least 1 CPU core that it can use. Due to the amount of resources available in the cluster, a reasonable solution was to partition all the Kafka topics to two partitions. This way, if we have only one executor for the application, it will read sequentially from two available partitions. When the workload for one executor becomes too high, we simply restart the application and assign two Spark executors to it. This way, both executors will belong to the same consumer group, but each one of them will, in parallel, read from different partition of the Kafka topic. Another parameter to set was replication factor for a Kafka topic. As stated in section 5.2, our Kafka cluster consists of three brokers, so the maximum replication factor we could set is three. On the other hand, *Zookeeper* [1] which is component required by Kafka to establish cluster coordination, requires a quorum of (n/2+1) instances up and running [16]. Since our Kafka-Zookeeper cluster consists of three nodes, quorum that Zookeeper requires is two. That means that our cluster can survive only one node failure at a time, which brought us to conclusion to set Kafka replication factor to two.
Spark also needs to be assigned enough resources in order to be able to consume messages that Kafka stores in its partitions. That is why we have conducted tests with different amounts of RAM memory and CPU, assigned to the driver and executor processes.

[1] https://zookeeper.apache.org/

id	devices amount	executor memory [MB]	number of executors	micro batch execution time [seconds]	elements in micro batch	mean execution time per element [milliseconds]
1	1000	1024	1	2	250	8
2	5000	1024	1	4	1250	3.2
3	10000	1024	1	5	2500	2
4	20000	1024	1	7	5000	1.4
5	35000	1024	1	12	8750	1.41
6	35000	1024	2	8	8750	0.941
7	50000	1024	2	11	12500	0.64
8	50000	2048	2	11	12500	0.64
9	70000	1024	2	14	17500	0.8

Table 6.1: Wibeee batch and speed layer tests

Another important configuration step needed to be performed is the way how Cassandra distributes data in the cluster. Cassandra offers two ways how hash function token range can be distributed - *single token range* or *vnodes* [17]. In order to easily scale the system, we configured Cassandra cluster to use *vnodes*. In that case, when new nodes enters the cluster, cluster balances itself by each node sending only small fraction of hash function segments that it stores to the new node, avoiding full data shuffle or deploying completely new cluster and migrating all the data, that *single token range* clusters cause.

6.2 Wibeee batch and speed layer tests

We have tested *wibeee pipeline* with different number of simulated devices that send data every minute, as it is the case with *wibeee* devices in the real world. We always had 1 core dedicated for the driver, with 512MB of memory as well as 1 core per executor. Results of the tests are available in the table 6.1. Experiments were ran with one executor until experiment number 5, where we saw that micro batch execution time exceeded 10 seconds; keep in mind that micro batch interval for both *wibeee* and *wattio* pipelines is 15 seconds. Later, for experiments 6 to 9, we were launching two executors, to read from two Kafka partitions in parallel. Increasing executor memory did not decrease the execution time of the process, as can be seen from experiments 7 and 8. That means that, when having 2 executors with each having 1GB of memory, it is enough to store data from 50000 devices in memory, without flushing outputs of data transformations to disk.
Also, on the figure 6.1, we see how much data Spark statefull transformation keeps in memory when experiment number 7 from table 6.1 was conducted. Since experiment 7 was conducted with two Spark executors, we see that each one of them stores around 12MB of data in memory. Comparing to available memory on each executor which is 1GB, we can bring conclusion that keeping Spark statefull transformation data in memory does not influence scalability of the process.
From the table 6.1, we see that with existing Kafka and Spark setup, the company can deal with up to 70000 *wibeee* devices. In experiment number 9, we see that 70000 devices are processed in under 15 seconds, which is micro batch interval for both pipelines. But, 14 seconds is a mean value, which can deviate,

Spark 1.6.1 Jobs Stages Storage Environment Executors Streaming

Executors (3)

Memory: 23.8 MB Used (2.6 GB Total)
Disk: 0.0 B Used

Executor ID	Address	RDD Blocks	Storage Memory	Disk Used	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time	Input	Shuffle Read	Shuffle Write	Logs	Thread Dump
0	ip-172-31-34-50.eu-west-1.compute.internal:49263	4	12.0 MB / 1247.3 MB	0.0 B	1	0	210	211	2.0 m	143.3 MB	410.5 KB	817.6 KB	stdout stderr	Thread Dump
1	ip-172-31-34-52.eu-west-1.compute.internal:57200	4	11.8 MB / 1247.3 MB	0.0 B	1	0	210	211	1.9 m	141.3 MB	404.2 KB	816.2 KB	stdout stderr	Thread Dump
driver	172.31.34.51:20487	0	0.0 B / 143.3 MB	0.0 B	0	0	0	0	0 ms	0.0 B	0.0 B	0.0 B		Thread Dump

Figure 6.1: Amount of data preserved in memory in *Wibeee* streaming process

id	devices amount	executor memory [MB]	number of executors	micro batch execution time [seconds]	elements in micro batch	mean execution time per element [milliseconds]
1	1000	1024	1	0.8	16	50
2	10000	1024	1	0.9	165	5.5
3	20000	1024	1	1	330	3.1
4	50000	1024	1	1.2	830	1.45

Table 6.2: Wattio batch and speed layer tests

due to lower performance of Amazon machine for example, and become higher than micro batch interval of 15 seconds. That would cause delays in start of the next micro batch interval, causing a *rolling effect* on all subsequent micro batches. Also, in case of crashes, Spark streaming recovers by reading *metadata* from S3 checkpoint directory, and processing data from the latest Kafka offset it has previously consumed, which causes number of elements in micro batch to increase. That is why, recommendation for Enerbyte team is to rely on this architecture up until they reach 40000 to 50000 *wibeee* devices, which is sufficient for Enerbytes foreseeable future.

6.3 Wattio batch and speed layer tests

Wattio sends much less data than wibeee (4 consumption data points in an hour instead of 60), so what we expect is *Wattio* data pipeline to be able to handle entire data load with only 1 Spark executor, for maximum number of *wattio* devices Enerbyte expects to have. Driver again had 512MB of memory and one core, and we were launching one core per executor. Tests in table 6.2 show the feasibility of the project. Wee see that in experiment number 4, we succeeded to process in real time 50000 *wattio* devices in under 2 seconds, which is as expected due to the fact that there are only 830 elements within 15 seconds micro batch interval (*wattio* device sends data reading every 15 minutes as opposed to *wibeee*, which sends data every minute).

6.4 Recommendations for higher workload

When Enerbyte exceeds 40000 to 50000 *wibeee* devices, they should consider following two alternatives:

- Pipeline algorithms are designed to be agnostic to a number of different projects (in *multitenant architectures* referred as *tenants*) data they process belongs to, which causes data from single micro batch to be inserted to different Cassandra keyspaces. When new project initiates, except creating new Cassandra keyspace, no settings need to be changed or Spark process to be restarted. On the other hand, this imposes a performance degradation, although we need to iterate over all the data from micro batch two times: first to get all the different projects data belongs, and then to filter all the data from the micro batch to a proper Cassandra keyspace. Since new projects are not frequently added and removed, pipelines can be modified in a way that list of projects (keyspaces) is provided through configuration file when Spark launch command is issued. This way, performance would be gradually improved, because we would remove one unnecessary iteration through entire micro batch data set.

- Introduce more partitions to Kafka topic, and launching Spark with same number of executors as there are topic partitions. That would introduce more parallelism, although data would be processed in more parallel manner. Maximum recommended number of Kafka topic partitions is twice the number of brokers in Kafka cluster [14]. Launching more that two executors on Kafka topic that has only two partitions is not recommended, because only two of them would be used, others would be idle; they would not perform any work, but would be reserved by the application and unavailable for the other Spark processes.

6.5 Platform costs

Since we have created Spark-Cassandra cluster with 3 Amazon *c3.xlarge* machines, as explained in section 5.2, total available resources in the cluster are 9 CPU cores and around 12GB of ram. *Wibeee* speed and batch layer when launched with 2 executors consume:

- 1 CPU core and 512MB of RAM memory for the driver;
- 2 CPU cores and 2GB of RAM memory for 2 executors;

Wattio speed and batch layer consume:

- 1 CPU core and 512MB of RAM memory for the driver;
- 1 CPU core and 1GB of RAM memory for 1 executor;

This means that total remaining amount of resources is 4 cores and 8GB of RAM memory, that Enerbyte team in the future can devote to running *wibeee* and *wattio* Spark batch processes, creating new data pipelines for new devices it starts to process data from or machine learning. Total price of the both Datastax as well as separate three node Kafka cluster, due to the fact that Enerbyte received Datastax Enterprise license for free, is 525 euros per month if all the instances are launched *on demand* or 400 euros per month if all the instances are reserved for one year with no up-front costs.

Chapter 7

Conclusion

In order for Enerbyte to be competitive on the market, they needed to develop a new data platform that can handle petabytes of data, that can scale and process data in real time, providing final users insights into their consumption as soon as the consumption has been achieved. New data platform definitively satisfies the needs of the company for the next foreseeable future, as well as enables them to grow bigger and develop new processes on top of it.
Advantage of using Datastax platform is also the ability to deploy Apache Solr [1], enterprise search platform on top of Cassandra, which is a use case Enerbyte might need in the future for other projects it plans to work on.
However, since these technologies are still immature, for a development of a simple data flow, lot of coding as well as lot of configuration needs to be done in order to obtain proper results and performance. When comparing ETL development that we are familiar with using ETL tools with easy to use graphical interface where we think more about the process logic than implementation steps, using Spark is definitively more difficult and challenging way to go. But, performance and benefits it provides are worth the effort of migrating to the solution like that.
Enerbyte is now competitive on the market and attractive to other businesses to establish cooperation with it. Platform that has been built will definitively strengthen Enerbyte's market position and increase the chances for achieving global business success.

[1] http://lucene.apache.org/solr/

Appendix A

Creating wibeee hourly data access object

BPMN diagram on figure A.1 conceptually presents logic of statefull transformation. *UpdateStateByKey* transformation is performed every micro batch interval (15 seconds), for both key - value pairs that are in the state (memory) and those that have just arrived for the first time. For example, if in the state we had keys 1,2 and 3, and in new micro batch keys 1 and 4, update state by key transformation will be executed for all of the following keys: 1,2,3 and 4. First branching in the BPMN diagram is based on whether new key has arrived to the system. If new key did not arrive, we check what is the state for that key. Business requirement is that if no new data for existing keys have arrived for 15 days, we release the key from the memory (we assume that device is no more functioning).
If new data for a specific key arrived, we check whether we have the state for that particular key. If we do not have the state, we set the state based on new data, and update Cassandra *wibeee_hourly* table. If we do have a state for newly arrived data, we perform state modifications. Results of statefull transformations are inserted into Cassandra.
On the diagram, you can see that there are multiple issues that are logged, so that process owners have full control of the data pipeline.

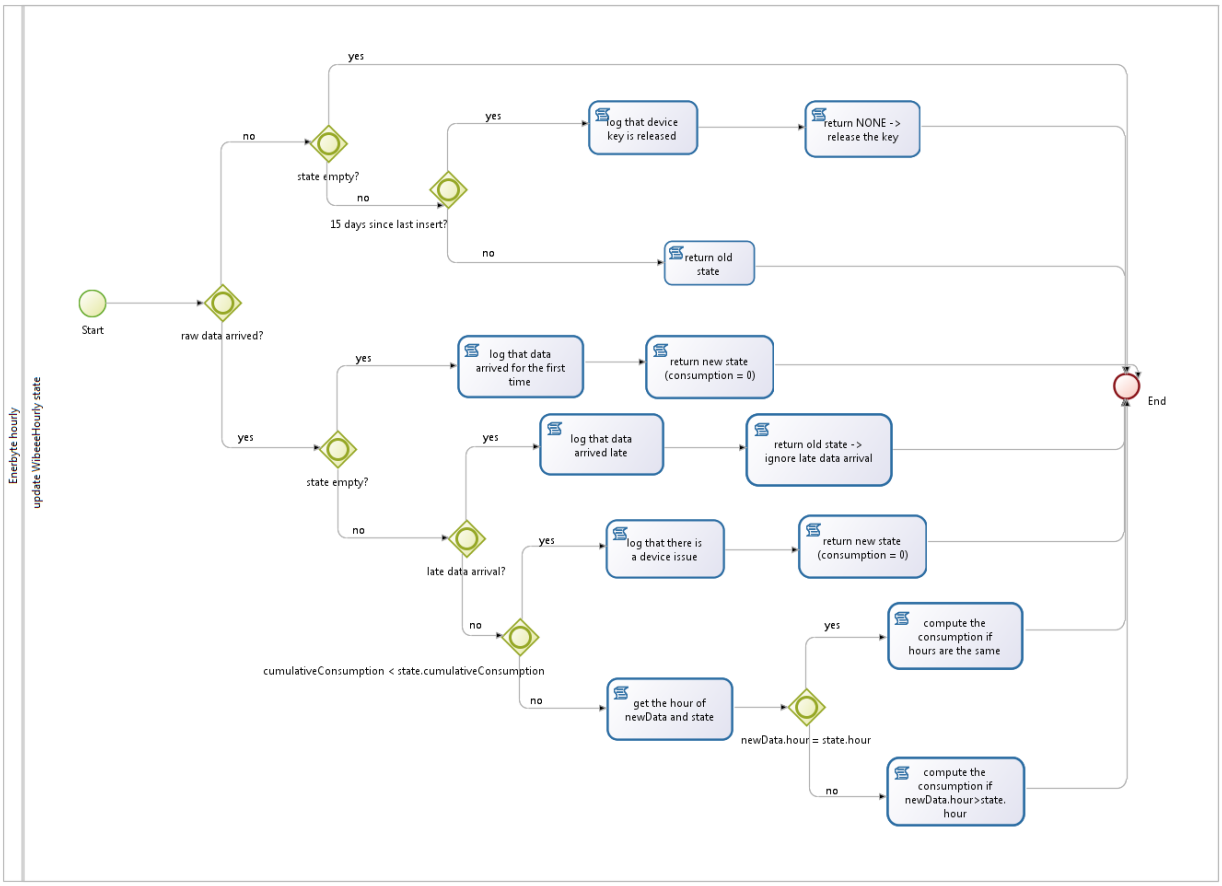

Figure A.1: BPMN Diagram: Creating Wibeee Hourly object

Appendix B

Wibeee Spark batch process logic

Figure B.1 conceptually represents logic of wibeee batch proces.
Firstly, process parameters are read from configuration file, after which, for each specified keyspace, data is

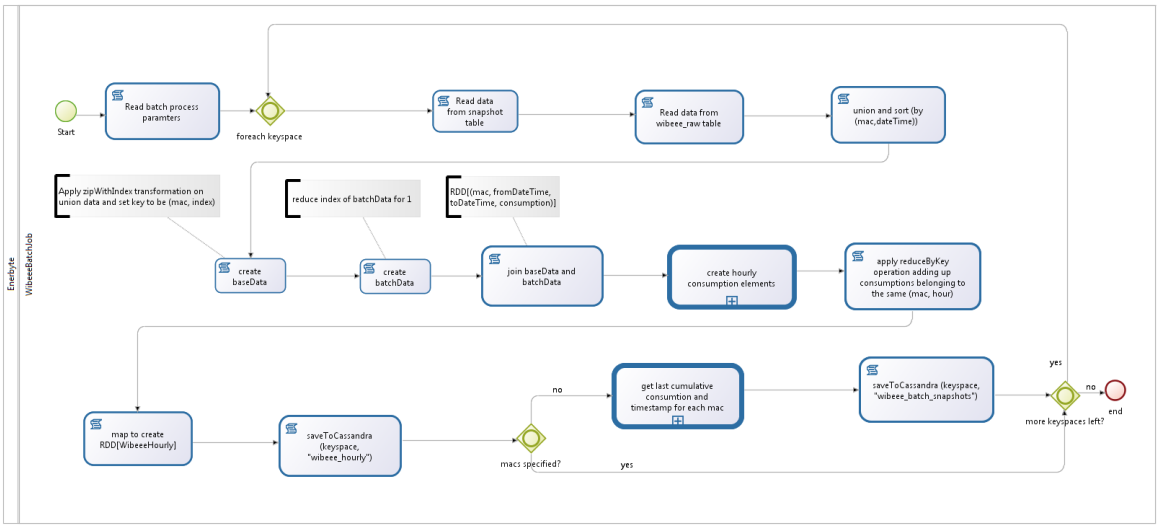

Figure B.1: BPMN Diagram: Wibeee batch logic

read from *wibeee_raw* and *wibeee_batch_snapshots* tables. Data is unioned, and sorted by (mac, dateTime). After that, by applying two Spark *zip with index* transformations, we create two datasets:

- *baseData*: Key-value data set with key *(mac, zipIndex)* and value *(dateTime, consumption)*. Example is available at figure B.2;
- *batchData*: Key-value data set obtained by reducing *zipIndex* of base data for 1. Example is available in figure B.3

baseData			
key		value	
mac	index	dateTime	consumption
1	1	05/05/16 06:59 AM	10
1	2	05/05/16 07:10 AM	11
1	3	05/05/16 07:34 AM	12
1	4	05/05/16 07:57 AM	13
1	5	05/05/16 07:59 AM	14
1	6	05/05/16 08:10 AM	25
1	7	05/05/16 08:55 AM	26
1	8	05/05/16 09:05 AM	36
1	9	05/05/16 09:34 AM	37
1	10	05/05/16 09:59 AM	38

Figure B.2: Base wibeee data

batchData			
key		value	
mac	index	dateTime	consumption
1	0	05/05/16 06:59 AM	10
1	1	05/05/16 07:10 AM	11
1	2	05/05/16 07:34 AM	12
1	3	05/05/16 07:57 AM	13
1	4	05/05/16 07:59 AM	14
1	5	05/05/16 08:10 AM	25
1	6	05/05/16 08:55 AM	26
1	7	05/05/16 09:05 AM	36
1	8	05/05/16 09:34 AM	37
1	9	05/05/16 09:59 AM	38

Figure B.3: Batch wibeee data

Base and Batch datasets are joined based on the key *(mac, zipIndex)*. Resulting dataset looks like on figure B.4. Consumption for all the matching keys is computed by subtracting value from batch data with corresponding value from base data.
In this model, every time interval between *fromDateTime* and *toDateTime* is considered to be a gap; only important thing is to determine whether the gap belongs to the same hour or not. When *fromDateTime* hour is equal to *toDateTime* hour, all the consumption is assigned to the hour both values refer to. When *toDateTime* hour is greater than the fromDateHour, we perform *linear interpolation* to resolve gaps between baseData and batchData values. After, we apply transformations to sum all the consumption values that belong to the same hour and save them to Cassandra *wibeee_hourly* table.
After storing computed values to *wibeee_hourly* table, only thing left is to update *wibeee_batch_snapshots* table, to insert the results of just finished batch process. For easier maintenance, we decided to keep the state

baseData joined with batchData				
key		value		
mac	index	fromDataTime	toDateTime	consumption
1	1	05/05/16 06:59 AM	05/05/16 07:10 AM	11-10 = 1
1	2	05/05/16 07:10 AM	05/05/16 07:34 AM	12-11 = 1
1	3	05/05/16 07:34 AM	05/05/16 07:57 AM	13-12 = 1
1	4	05/05/16 07:57 AM	05/05/16 07:59 AM	14-13 = 1
1	5	05/05/16 07:59 AM	05/05/16 08:10 AM	25-14 = 11
1	6	05/05/16 08:10 AM	05/05/16 08:55 AM	26-25 = 1
1	7	05/05/16 08:55 AM	05/05/16 09:05 AM	36-26 = 10
1	8	05/05/16 09:05 AM	05/05/16 09:34 AM	37-36 = 1
1	9	05/05/16 09:34 AM	05/05/16 09:59 AM	38-37 = 1

Figure B.4: Base and Batch datasets joined

only in case when batch process is performed for all the devices in the system. When that is the case, for each mac that was processed, we take the last cumulative consumption that was processed, and put it into the *wibeee_batch_snapshots* table. Next time data engineers want to perform batch process, *start_date_hour* needs to be set to highest *end_batch_date_hour* from the snapshots table.

Appendix C

Wattio Spark batch process logic

Logic used to implement wattio batch process is best summarized in figure C.1 For each keyspace, data

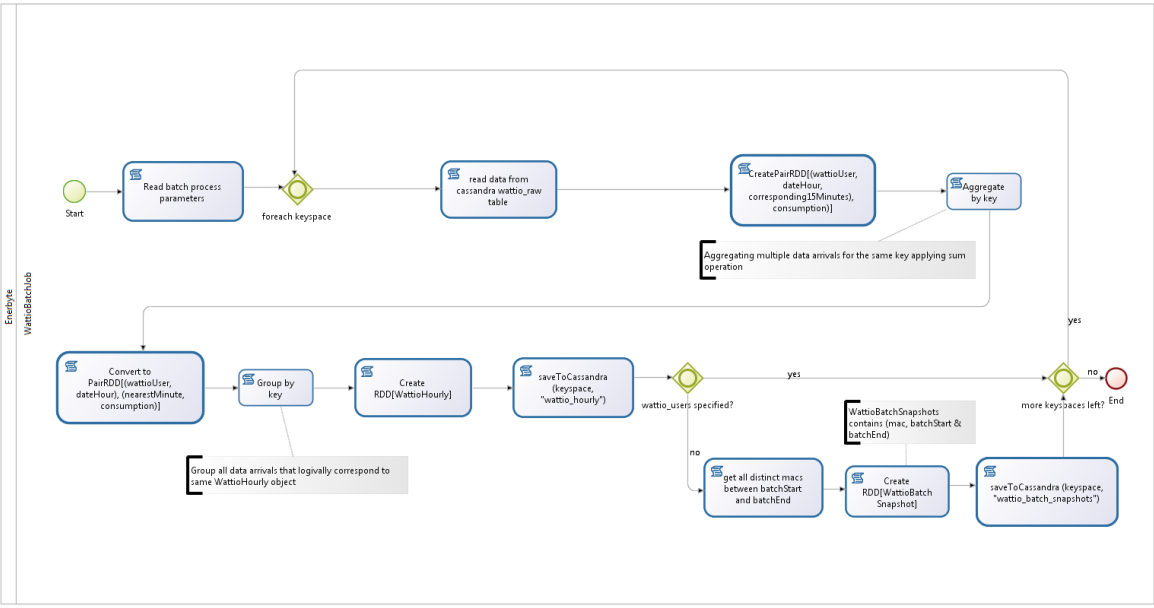

Figure C.1: BPMN Diagram: Wattio batch logic

is being read from Cassandra table *wattio_raw*, after which key value data structure is created with key *(wattioUserId, dateHour, floor15Minutes)* and value *consumption* and is aggregated by key. This transformation, resolves possible two data arrivals within same 15 minutes interval. Enerbyte has never experienced a problem like this, but due to Company expansion and more devices that are being processes, this kind of error might occur in the future. After, new key value data structure with key *(wattioUserId, dateHour)* and value *(floor15Minutes, consumption)* is created and transformation *groupByKey* is performed, which groups

all data points belonging to the same user and hour. After, *WattioHourly* data access object for *watio_hourly* Cassandra table is created, and data is inserted into Cassandra.
Same as for wibeee, we update *wattio_batch_snapshots* table only if batch process was performed for all the users available in the system.

Bibliography

[1] R. Pérez-Chacón, R. L. Talavera-Llames, F. Martínez-Álvarez, and A. T. Lora, "Finding electric energy consumption patterns in big time series data," in *Distributed Computing and Artificial Intelligence, 13th International Conference, DCAI 2016, Sevilla, Spain, 1-3 June, 2016*, pp. 231–238, 2016.

[2] "Service oriented architecture principles." http://serviceorientation.com.

[3] "Spark cassandra connector documentation." https://github.com/datastax/spark-cassandra-connector.

[4] "Kafka and spark integration guide." http://spark.apache.org/docs/latest/streaming-kafka-integration.html.

[5] N. Marz and J. Warren, *Big Data: Principles and best practices of scalable realtime data systems*. MANNING, April 2015.

[6] "Choosing the best serialization format for kafka." http://www.confluent.io/blog/stream-data-platform-2/.

[7] "Time series modeling in cassandra." https://academy.datastax.com/resources/getting-started-time-series-data-modeling.

[8] "Linear interpolation." https://en.wikipedia.org/wiki/Linear_interpolation.

[9] "Mesos and marathon." https://github.com/mesosphere/marathon.

[10] "Dc/os architecture." https://docs.mesosphere.com/1.7/overview/concepts/.

[11] "Dc/os recommended deployment strategies." https://www.digitalocean.com/community/tutorials/how-to-configure-a-production-ready-mesosphere-cluster-on-ubuntu-14-04.

[12] "Dc/os bug report portal." https://dcosjira.atlassian.net/browse/DCOS-131.

[13] "Spark deployment mode in cluster." http://spark.apache.org/docs/latest/cluster-overview.html.

[14] "Choosing proper number of kafka partitions." http://www.confluent.io/blog/how-to-choose-the-number-of-topicspartitions-in-a-kafka-cluster/.

[15] "Kafka partitions consumed by spark executors." http://stackoverflow.com/questions/37810709/kafka-topic-partitions-to-spark-streaming.

[16] "Kafka and zookeeper installation." http://stackoverflow.com/questions/37861050/installing-kafka-cluster.

[17] "Single token range vs vnodes in cassandra." http://www.datastax.com/dev/blog/virtual-nodes-in-cassandra-1-2.