UiO : Department of Informatics
University of Oslo

An open database model and Android database application for storing Obstructive Sleep Apnea data

Student:
Viet Thi Tran

Supervisor:
Prof. Dr. Thomas Peter Plagemann

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

in the

Distributed MultiMedia Systems (DMMS)
Department of Informatics

April 29, 2017

Abstract

The Faculty of Mathematics and Natural Sciences
Department of Informatics

Master of Science

An open database model and Android database application for storing Obstructive Sleep Apnea data

by Viet Thi Tran

Sleep apnea is a sleep disorder that causes repeatedly reduced respiration or no airflow in a period of time during sleep. Sleep apnea can be detected by analyzing bio-physiological signals such as electrocardiogram, oxygen saturation, and respiratory effort, etc. The bio-physiological signals that are recorded during sleep, are usually kept in files which have low level abstractions, indexing primitives, and diverse formats such as EDF/EDF+ file formats, or a Waveform Database format (PhysioBank databases). To manage and query data from these files the provided software packages are required, i.e. the WFDB software package for the Waveform Database format. Otherwise, it is necessary to write a new software application based on the specification of the file formats, which leads to increased difficulty of programming and analyzing bio-physiological data. The thesis presents design and implementation of a relational database model for storing not only Obstructive Sleep Apnea signals, but also other bio-physiological signals. The first benefit of using a relational database for storing bio-physiological data is that there is no need to write or include management tools for managing the collected data. Furthermore, data analysis can be directly performed on the collector devices (mobile devices) by using the SQL language, which provides many useful algorithms for analyzing data. In addition, remote services can ask for some parts of data from collector devices by using remote querying. Last but not least, the privacy of patients is not violated if they can keep their bio-physiology on their own devices, and can decide which data they would like to send. The database model that is presented is this thesis is a platform independent, it can therefore be implemented on whatever database management system as long as they support SQL language. In this thesis, a design and implementation of a database application for the database model are proposed. The design is discussed first at an abstract and platform independent level, in which it describes the ways the database application works with real-time sources and files. The Android platform is chosen as implementation platform for both the database model and database application. Data sources for the database are from the CESAR acquisition tool and EDF files that are exported from PhysioBank.

The database size is less than 1736 MB after collecting data from all channels of the CESAR acquisition tool with 100Hz for each channel in nearly 19 hours. That is, the application can collect data for a whole day without any storage problem. The power consumption for the application is 6.6% of the total power drain on the device, which is quiet efficient. The average read time and write time are 17936 samples per second and 12960 samples per second. With a stress test, the application shows that it can manage all channels of 14 BITalino sensor kit with 100Hz while importing a EDF file, exporting data from six channels to a EDF file, and visualizing incoming samples on a graph.

Nevertheless, the results show that the database model and database application are very efficient for storing and analyzing Obstructive Sleep Apnea signals, and other bio-physiological signals.

Acknowledgements

First of all, I would like to thank my supervisor Professor Dr. Thomas Plagemann, who has conveyed a spirit of adventure in regard to scholarship and research to me, and provided invaluable insight through my thesis. Without his support and guidance, this thesis would not have been possible.

I would also like to thank my Norwegian teacher Marit Ruud McGhie, who constantly encouraged me in my early days in Norway. Without her encouragements, I could not pass the Norwegian course, and this thesis therefore might be done by someone else.

My sincere thanks also goes to the staffs and students at the Department of Informatics who contribute a friendly environment at the faculty, and thanks to my friends who have constantly blessed me in my daily work.

Special thanks to my family: my parents Tran Van Huu and Ngo Thi Xuan Huong for giving birth to me at the first place and supporting me spiritually throughout my life. I would also like to thank my brothers and sisters for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of writing this thesis.

Last but not the least, I would like to thank my partner Vo Thi Hanh Nhan, who has been trusting and waiting for me for nearly 10 years. It's time to end the wating time.

Absence diminishes small loves and increases great ones, as the wind blows out the candle and fans the bonfire.

François de La Rochefoucauld

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Background and Motivation

Sleep apnea is a sleep disorder that happens when breathing is interrupted during sleep. The interruption is caused by either the soft tissue in the back of the throat that collapses and blocks the airway, or the breathing muscles failed to receive signals from brain, or both. Interruption in breathing leads to lack of oxygen to the brain and the rest of body, which causes many health problems such as high blood pressure, heart failure, irregular heartbeats, headaches, etc. One of sixth adults in Norway are suffering from Obstructive Sleep Apnea in a report from helsenorge.no (Helsenorge.no, 2014), and another report illustrates that one fifth of the US adults are affected from the illness, and about 90% of all US adults are undiagnosed (Sleep_Medicine, 2009).

To determine sleep disorders, a polysomnogram is considered the gold-standard for diagnosing the illness. A polysomnogram is both a diagnostic tool, and research tool that is used for determining sleep disorders. However, the polysomnogram requires a patient to stay overnight at a sleep laboratory where the patient is monitored by qualified personnel (Naresh, 2008). This procedure is inflexible, time consuming, and costly. With polysomnogram, the collected data are stored on a stationary machine at the clinic, hence, it is difficult for patients to access their data in case they want to share the collected data, i.e., to another physician or clinic. Beside the polysomnogram, there are many wearable sensor kits that can be used for collecting bio-physiological samples. By using these kits, patients do not need to stay overnight at a clinic to be monitored during their sleep. They can stay at home, collect samples, then send the data to physicians after the collection process is finished. Currently, the wearable sensor kits that can be used for diagnosis at home can be divided into two types. The first type is certified sensor kits such as NOX-T3, NOX-A1, etc., that usually include analysis software, user manuals, etc. However, devices of this type are usually very expensive, a NOX-T3 sensor kits with software and user manuals from nox-medical costs 55000kr plus value added tax (Thomas, August 26th 2016). Another type are consumer electronic sensor kits such as BITalino, that often come with reasonable or even low price, because the quality of signals have not been approved, or maybe they are not as good as the certified sensor kits. Moreover, analysis

softwares are rarely included, users have to buy the related softwares, or have to implement analysis applications based on sensor kits specifications in order to use the sensor kits. At the time of this writing, the bio-physiological signals that are collected from these kits are kept in files which have low level abstraction. The level of abstraction can be divided into two levels, which are low level abstraction and high level abstraction. A high level abstraction provides a flexible way to interact with data files, and a user does not need to know in detail how the data are organized in other to use the data files. In contrast, a low level abstraction is more detail, and if a file is saved as low level abstraction, an application must follow exactly the file description in order to read the file. If the application does not follow some steps in the description, the application fails to read the file. A good example for this explanation is an EDF file from NOX-T3. The file can not be read by a famous EDF parser, which is EDFbrowser (teuniz.net, 2017), since the file is not saved as the EDF specification. That is, the file is specified as a EDF plus file, but it does not contain an Annotation channel. Moreover, to read and search on a file with low level abstraction requires a good programming skill to write functions for each action on the data file. An entire recording is usually kept in a file on the collector devices, i.e., a telephone or a table. The whole file could be forwarded to an analysis station or a clinic. This is inefficient and cumbersome if the patients are required to only send some part of the record.

Today, mobile devices are popular, and the processing capacity and storage space are constantly increasing while the price is decreasing monthly. Devices that are powered by Android and iOS can be used as a collector, which receives and stores data from sensors, for the wearable kits. To be easier to find, export, analyze, etc., other data types can be included to the collected data such as timestamp, descriptions, etc. The Android and iOS operating systems integrate the SQLite database management system which allows to store the bio-physiological signals in a relational database. By using SQL queries, users can easily choose samples from the channels, i.e., ECG, EEG, etc., they want to export and send to physicians. Moreover, data analysis can also be performed directly on mobile devices, and only results from the analyses can be sent to the physicians. Nevertheless, mobile devices need to be connected to some computers to store or forward the sensor data because of the limitation of not only storage capacity, but also the data synthesis of the mobile devices.

1.2 Problem Statement

Bio-physiology signals are usually stored in different formats which are defined by different clinics. In other words, each clinic has each own format to store and manipulate bio-signals. Therefore, to use the collected data outside of the clinic, the corresponding software packages, or at least the format specifications are needed. Moreover, an analysis function needs to be written for each data analysis, because most of the formats for storing bio-physiology signals are defined as low abstraction

level such as files with indexing primitive. Hence, the difficulties of programming and analyzing bio-physiological data are increasing. Therefore, the goal of this thesis is to develop an alternative solution for storing bio-physiological signals in addition to the existing solution, i.e., the EDF/EDF plus file formats. The provided solution in this thesis is not only opening to store all bio-physiological data, but also to simplify easy data analysis. This can be achieved by defining a relational database model and a database application. Hence, bio-physiological data can be translated into tables and relations, which can be easily analyzed by leveraging the advantages of SQL, and are easy to extent by adding new tables and relations. However, defining a relational database model that is open to store all bio-physiological data, and creating a database application that works as a wrapper for all data sources is challenging. This is because the diverse formats of data sources causes difficulties to define an universal interface, which can support not only importing data from current data sources, but also being open for future data sources. In addition, the challenge also comes from the limitation in hardware of a mobile device. The storage space of a mobile device must not be wasted and must not be used as a permanent storage to keep all collected records. Hence, a design must take the overhead of the metadata, and data backup into consideration.
All collected OSA data signals from sensor kits must be kept in an efficient way such that the collected data can be easily retrieved and used. If the data formats provided by the CESAR acquisition tool(Gjørby, 2016) and the EDF/EDF plus are still used to store the collected data, it is difficult to retrieve and use the collected data. Therefore, a relational database model and a database application are provided in the thesis to improve the process of collecting and processing OSA data signals. The database model and database application are also open for other bio-physiology data signals.

1.3 Contributions

Today, most of the mobile devices are powered by an operating system such as Android or iOS that integrates a database management system, like SQLite. Therefore, this thesis proposes an open database model and a database application for storing Obstructive Sleep Apnea monitoring data and other bio-physiology signals on an Android device. The thesis defines and analyzes requirements for a relational database model that can be used for not only storing data from the CESAR acquisition tool(Gjørby, 2016) and EDF/EDF plus files, but also opening for other bio-physiological data. As a result, a platform independent database model is proposed, followed by a physical database model for the SQLite database management system which is integrated in the Android operative system. The thesis also designs and develops a database application, in which two sensor wrappers are provided to import data from the CESAR acquisition tool and the EDF/EDF plus files. The

database application also allows users to perform data analysis by using SQL queries, or by visualizing the collected data on a graphical view.

1.4 Structure of the Thesis

Chapter 1 presents an overview over the motivations for the thesis, the problems the thesis has to deal with, and the contributions of this thesis. Chapter 2 provides a general discussion of the Sleep Apnea illness, in which a taxonomy of the illness and observable characteristics of Obstructive Sleep Apnea are presented. Data sources that are used in this thesis are discussed in Chapter 3. There are two types of sensor sources that are presented in the chapter; a real time source which is the BITalino sensor platform, and a non-real time source which are the databases from Physionet. Specifications for these sources are also presented in this chapter, and they are the foundation for the later chapters. A database model for Obstructive Sleep Apnea monitoring data and other bio-physiological data is introduced in Chapter 4. In this chapter, requirements for the database model design are carefully discussed, then step by step the database modeling is presented. Results from Chapter 4 are a platform independent database model (logical model) and a specific database model for SQLite (physical model) that are implemented in the next chapter. Chapter 5 proposes a database application for the designed database model in Chapter 4. At first, a database application design is provided. The design can be implemented on an Android or iOS devices, or even on a personal computer. The Android operating system is chosen as an implementation platform for the design in the last section of the chapter. The efficient of the database design and database application is evaluated in Chapter 6 by performing different experiments on the database application. Finally, a synthesis is made, open problems are discussed, and future works are addressed in Chapter 7.

Chapter 2

Sleep Apnea

Sleep apnea is a disorder where breathing stops and starts repeatedly during sleep. Stop breathing happens when either the airway collapses or the brain can not successfully send the signal to the breathing muscles. The first case is called Obstructive Sleep Apnea (OSA), and the second case is called Central Sleep Apnea (CSA). In both cases no air come in or out of the lung. It often lasts from a few seconds to minutes, and can happen about 30 times or more per hour(nhlbi, May 2009). Sleep apnea is difficult to diagnose, because it happens when the patient sleeps. Moreover, sleep apnea can not be detected by normal tests such as blood test. It can be noticed by a family member when the patient intensively snores. However, CSA does not often come with snoring, therefore there it is difficult to detect sleep apnea in general. Sleep apnea can lead to many serious diseases if it is untreated, such as high blood pressure, heart attack, stroke, heart failure, etc. Getting not enough sleep can cause serious accidents if truck or taxi drivers have this disease. Section 2.1 presents a taxonomy of sleep apnea. Section 2.2 describes the physiological signals that can be used to diagnose sleep apnea.

2.1 Taxonomy of sleep apnea

Sleep apnea is more dangerous than people have thought, because it can lead to many other well-known diseases. According to The Akershus Sleep Apnea Project, there are 25% middle-aged Norwegians at high-risk of having OSA(Harald Hrubos-Strøm, 16 June 2010). That means sleep apnea is very common and it has to be studied seriously. In fact, there are dozen of research papers about it. When searching with "sleep apnea" on Google Scholar, it returns about about 18900 related articles. Sleep apnea is divided into three types:

- OSA occurs when the soft tissue falls to the back of the throat which causes the breathing to be obstructed during sleep.
- CSA occurs when the brain fail to sends the signals to the breathing muscles and therefore it fails to control the breathing cycle.
- Mixed sleep apnea (MSA) is a combination of both OSA and CSA.

2.1.1 Central sleep apnea

"Central sleep apnea syndrome is characterized by a cessation or decrease of ventilatory effort during sleep and is usually associated with oxygen desaturation."(American_Academy_of_Sle 2005)

CSA happens when the brain can not send the appropriate signals to muscles which control breathing. CSA is not as common as obstructive sleep apnea. CSA is sometime a consequence of other diseases like heart failure, stroke, or sleep at high altitude.

Symptoms: Common symptoms of CSA are awakening suddenly with breathlessnesses, insomnia, headache in the morning, difficulties to focus on work, hypersomnia, or snoring. The bed-partner can notice that the patient may stop breathing a number of times during sleep, about 30 times or more per hour, and the patient may snore. However, snoring is not common in CSA, and snoring does not always mean that a patient has sleep apnea.

Causes: There are many causes that make the brain fail to connect with the breathing muscles. One of them is Cheney-Stokes, which is often associated with congestive heart failure or stroke, and is characterized by rhythmic cycles gradually increased and then decreased breathing which results in a temporary pause in breathing. Other reasons which cause central sleep apnea are certain medical conditions. For example, if a patient has problem in cerebrovascular or has brain tumors can cause that signals to the breathing system get lost. Certain drugs such as opium, morphine or codeine can lead to irregular breathing like increase, decrease or stop breathing. Sometime sleep at high altitude can cause central sleep apnea, but it is not a big problem, because the problem will disappear when moving to a lower altitude.

Complications: Not getting enough sleep can cause cardiovascular diseases. When the oxygen level in the blood is low, the heart will work hard to deliver blood to the tissues. If this lasts for a long time it may cause heart failure. Likewise, when the heart works hard, the pressure on the vascular increases which can cause hypertension. Hypertension is one of the most dangerous diseases and can lead to stroke or even dead. Waking up many times at night will make us tired and difficult to restore normal sleep. People who do not get enough sleep often have severe daytime sleepiness, fatigue and irritability. They can not focus on work and sometime fall asleep, not only at work but also when they drive a car.

Diagnostic methods: Usually a doctor may perform an evaluation based on those symptoms which are mentioned above, or a patient must be sent to a treatment center for sleep disorders. At the treatment center, the patient can be analyzed by overnight monitoring breathing and other body functions during sleep. In polysomnography

testing, the patient is connected to monitoring devices such as heart, lung, brain activity, breathing patterns, arm and leg movement, blood oxygen during levels sleep. Moreover, a review by a cardiologist or a doctor who specializes in the nervous system may also necessary to find the causes of central sleep apnea.

Treatments: The first step of treatment is to try to cure the other diseases which cause the central sleep apnea. For example, a good treatment of a heart failure may eliminate central sleep apnea. The other way to treat this disease is to use drugs. There are some drugs which have been used to stimulate breathing for those who have central sleep apnea. For example, acetazolamide can be used to prevent central sleep apnea at high altitude. However, other drugs can cause apnea, for example opioid. The dose of opioid drugs need to be reduced if it causes central sleep apnea. However, drugs are not the major solution to treat central sleep apnea.
There are four physical treatment methods which are often used in treating central sleep apnea. The first one is continuous positive airway pressure (CPAP), which uses mild air pressure to keep the airway open. This treatment is mainly used for obstructive sleep apnea where the patient wears a mask on the nose during sleep, but it is also used for central sleep apnea. Another useful method is bilevel positive airway pressure (BPAP) which supplies stable and continuous pressure in the upper airways while breathing in and out. BPAP provides high pressure when inhaling and low pressure when exhaling. The purpose of this treatment is to boost the weak breathing pattern of central sleep apnea into normal. Some BPAP devices can detect if there is no air-follow in a few seconds it will automatically active the breathing system. There is a better solution than CPAP and BPAP, that is called adaptive servo ventilation (ASV). ASV is designed to treat central sleep apnea and complex sleep apnea by monitoring normal breathing pattern and storing it in a database system as a training set. When the patient falls asleep, ASV monitors the breathing process and compares it with the training set. If there is apnea, ASV will use pressure to regulate the breathing pattern and prevent pause in breathing. The last method is using supplemental oxygen, but there are many arguments against this method. The main argument is that the oxygen level may normalize, but the carbon dioxide can not release. As a result, a signal will be sent to brain which causes awakenings and the sleep will be fragmented.

2.1.2 Obstructive sleep apnea

"Obstructive sleep apnea syndrome is characterized by repetitive episodes of upper airway obstruction that occur during sleep, usually associated with a reduction in blood oxygen saturation."(American_Academy_of_Sleep_Medicine, 2005)

Several types of sleep apnea exist, but the most serious and common type is obstructive sleep apnea. It occurs when the throat muscles relax and blocks the airway during sleep. The most noticeable signal of obstructive sleep apnea is snoring, although not everyone who snores has obstructive sleep apnea. It usually affects old

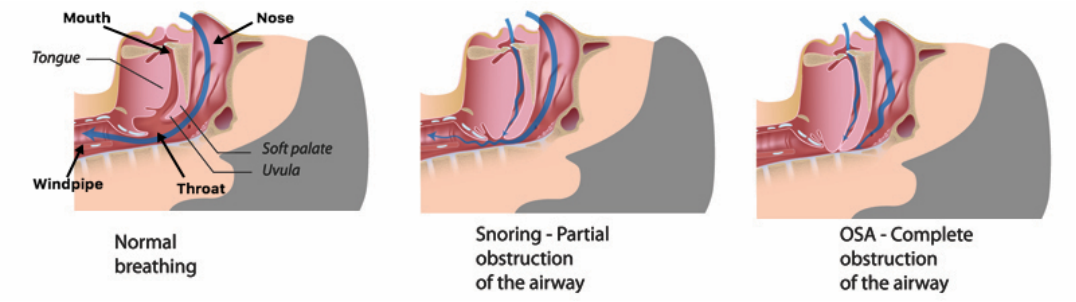


FIGURE 2.1: Obstructive Sleep apnea (sleeptmjdr.com, 2016)

people, but anyone can have this illness. Obstructive sleep apnea is particularly common for those who are overweight. We discuss more in detail about its symptoms, causes, complications, diagnostic methods, and treatments below.

Symptoms: As mentioned above, the most common signs is loud snoring and sometimes pausing in snoring. When there is a pausing in snoring, choking and gasping may follow it. Apnea occurs when the throat muscles relax and block the airway, therefore when we sleep on our back the snoring will be loudest. When we turn on our side, the snoring sound will decrease or even stop. That explains why we do not snore every night. It is difficult for us to recognize that we are snoring. However, a family member can easily notice the problem. Another sign is that we have excessive sleepiness during the day, can fall asleep while working, watching television or even driving. When we do not have normal breathing at night, our brain does not have enough oxygen to work. It causes not only morning headaches, but also memory problems. The other good sign to detect obstructive sleep apnea is waking up with dry mouth or sore throat.

Causes: The airway composes of mouth, nose, throat, and windpipe. It always opens under sleep when we do not have sleep apnea.

However, the airway becomes narrow when the soft palate and uvula relax. As a result, it is difficult to breath, and the patient might snore. This is called partial obstruction of the airway. When the muscles in the back of the throat relax too much, they will block the airway, as a result normal breathing is impossible. This situation is called complete obstruction of the airway, or obstructive sleep apnea. Blocked in breathing may cause a lower oxygen level in blood. Hence, the brain sends a signal to wake up the patient that the airway can be reopened. This process usually happens in a very short time and it is often not noticed because shortness of breath may be fixed with one or two deep breaths.

Major causes of obstructive sleep apnea are old age, brain injury, decreased muscle tone, increased soft tissue around the airway, obese, or structural features that give rise to a narrowed airway(Wikipedia, 2016). Everyone can have obstructive sleep apnea, but certain factors imply a higher risk. The most significant factor is excess weight. Over a half of the people with OSA are overweight, because the fat of the upper respiratory can block breathing. The size of the neck may indicate whether someone at risk of OSA or not, because a thick neck can narrow the airway. Other

risk factors are alcohol and smoking. Mihaela Trenchea et.al have researched on this topic and showed that smokers who smoke more than one pack of cigarettes a day are at higher risk of severe obstructive sleep apnea than those who do not smoke(Mihaela, 4.2016). Other minor but not least causes are sex, age, genetic, or race. Overall, men are twice likely to develop OSA than women, and older have this illness more often than younger.

Complications: Obstructive sleep apnea is a serious health problem which leads to several complications like:

Cardiovascular diseases. As mentioned in central sleep apnea complications, when the tissues do not get enough oxygen, the heart will beat fast and hard. As a result, the pressure on the heart and vascular will increase. About half of people with sleep apnea have high blood pressure, which increases the risk of heart failure and stroke. The more serious obstructive sleep apnea, the higher the risk of high blood pressure. With an underlying heart disease, a repeated low blood oxygen level can lead to sudden death from a cardiac event. Moreover, people with obstructive sleep apnea are often capable of developing abnormal heart rhythms like atrial fibrillation.

Medications and surgery. In general anesthesia, the doctor will use some medications like narcotic analgesics which can relax the patient's upper airway. When patient lies on his back, it will cause severe problems. Therefore it is important that patients with obstructive sleep apnea or related symptoms must inform the physician.

Tired and sleepiness at daytime. Similar to central sleep apnea, daytime drowsiness, fatigue and irritability are results from repeatedly awaking at night. Moreover, people with obstructive sleep apnea have often difficulties to focus on work or driving. Hence, it is not only dangerous for the patient, but also dangerous for people who are around him.

Partner's insomnia. Snoring may be a huge problem for those who have to listen to it. Sleeping with a snoring bed partner sometime is a nightmare, because the people who do not snore can be woken up by the snoring sound. As a result, the partner may get some health problem related to insomnia. In some cases, the partner may leave sharing room to find a quiet place to sleep, or even it is a cause for divorce.

Diagnostic methods: The evaluation may usually be performed by observing signs and symptoms. Then some tests need to be done, such as measuring the neck or checking the blood pressure. The medical doctor may examine the back of the throat, mouth and nose to find out if there are any abnormalities. Further evaluation requires to stay overnight at a sleep center. At that place, breath and other body functions are monitored while the patient is sleeping.

Sleep tests. A patient may be asked to stay at a clinic for monitoring the heart, lung, brain activity, body movements, breathing patterns, and blood oxygen levels during sleep by using professional devices. Based on the information from those sensors, the physician can diagnose obstructive sleep apnea and its level of severeness. Sometimes a patient has other diseases which also have the same symptoms as obstructive sleep apnea. For example, narcolepsy also causes excessive daytime sleepiness, but does not have the same treatment.

Measuring oxygen. The patient does not need to come to a sleep center. Instead, a small sensor which can measure the oxygen level in blood can be used. The patient can stay at home and wear this sensor on his finger to carry out the measurement. It is very simple, consistent and painless. While the patient sleeps, the sensor collects the information about oxygen level, and may store it on a computer or smart phone. Then the data will be sent to the physician. Based on the pattern of oxygen level in blood, the physician can determine if the patient has obstructive sleep apnea, or the patient may be recommended a sleep test at a sleep center.

Otolaryngology tests. This is often performed when a patient snores too loud. As mentioned earlier, when someone snores, it does not mean that he has obstructive sleep apnea. However, over a half people who snore have this illness. Hence, examinations on the nose, mouth, throat, palate, and neck need to be carried out.

Cardiopulmonary tests. These tests usually relate to the measurements of the air flow, the breath samples and the heart rate samples. By observing the heart rate and respiration from the abdomen and chest, the physician can determine whether the patient has obstructive sleep apnea or not.

Treatments: We can apply three types of treatments which are lifestyle changes, therapies, and surgery or other procedures (mayoclinic.org, June 15, 2016).

Lifestyle changes. This is the first treatment to try when OSA is diagnosed. Half of people who have obstructive problem are obesity, hence trying to loose weight is the first thing to do for the overweighted patients. Beside weight-loss, avoiding alcohol is helpful, since alcohol can also cause this illness. Therefore, one should not drink too much alcohol, especially several hours before bed (mayoclinic.org, June 15, 2016). Obstructive problem is happened when the airway at the throat collapses. Therefore, it is better to lie on the side instead of lying on the back.

Therapies. The methods applied for central sleep apnea are also applied for OSA, normally are CPAP, BPAP and ASV.

Surgery. The goal of surgery is to remove the excess tissue from the throat which can collapse and block the airway. There are several surgical options: surgical opening in the neck, surgical removal of tissue, jaw surgery, upper airway stimulation, and

implants (mayoclinic.org, June 15, 2016).

2.1.3 Mixed sleep apnea

Mixed sleep apnea is also known as complex sleep apnea. It is a merger of central sleep apnea and obstructive sleep apnea. It is identified by researchers at Mayo Clinic (sciencedaily.com, September 4, 2006). Mixed sleep apnea is caused when the respiratory control is interfered. Namely, the brain sends a signal to control the breathing system, but the responses of the breathing are not as the brain desired. When using CPAP, it can cause chaos on the ventilator control. As a result, obstructive sleep apnea can be cured, but the central sleep apnea can occur because the signals which are sent to the breathing system do not work. New CPAP devices can supply additional carbon dioxide to stabilize the breathing pattern to avoid mixed sleep apnea (Muhammad, February 16, 2014).

2.2 Observable characteristics of obstructive sleep apnea

By examining the sensors' abilities of observing the signals which can be used to observe obstructive sleep apnea, we suggest to divide those signals into two categories. The first group comprises signals that can be simply measured by sensors which are integrated in smart wearable devices. The other group includes complex signals which can only be obtained by professional devices and can be transfered to user devices. Simple signals are the heartbeat, the change the volume of the chest and abdomen, and snoring. The complex signals include EEG, EMG, EOG, ECG, oxygen saturation in the blood, breathing, and blood pressure. Some non-professional devices can also obtain these signals. However, the results from them are not good enough to use for medical diagnosis. For instance, the BITalino can observe almost all of the complex signals, but it is not a medical device as disclaimer on the BITalino website.

Heartbeat. The normal heart rate of a person varies from 60 to 100 beats per minute (heart.org, April, 2016) depending on how often that person trained. Hence, each person has a stable heart rate in this range. However, sleep apnea causes the heart rate to increase and to break the ordinary pattern. Heart rate is usually easily measured by using a smart watch or smart ring, or even a tattoo circuit. These devices can collect data and may send it to a smart phone to analyze it.

Snore. Snoring can be measured by the microphone of a smart phone. The received signal is compared with available data samples. The result of the comparison can be used to determine whether the snoring is related to OSA or not. Snoring is often stopped after a period of time when people awake and start snoring again.

Measure the volume of air to breathe. When observing the movement of the chest with a camera phone, Reyes gave Bersain(Bersain Reyes, February 25, 2016) can calculate the average respiratory rate and tide volume.

In addition to the signals which can be measured by integrated sensors on the smart devices, we can keep track of OSA by using non-integrated sensors to measure

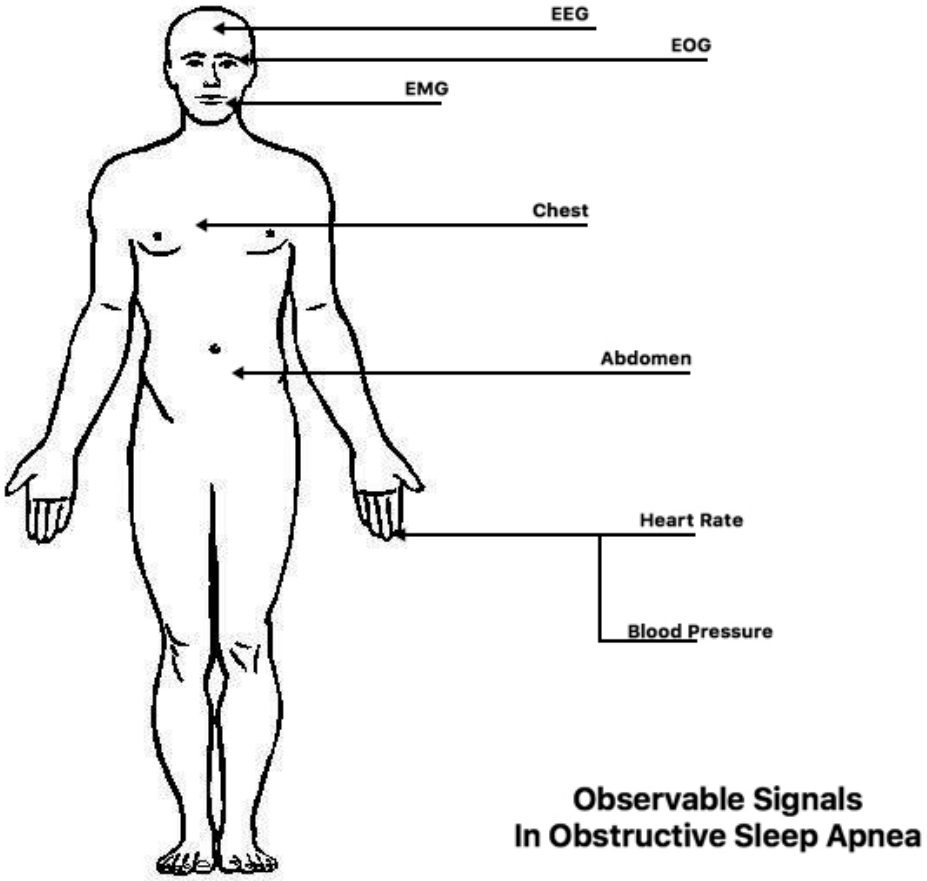


FIGURE 2.2: Observable Signals (heart.org, June 2001)

signals such as electroencephalogram (EEG), electromyography (EMG), electrooculography (EOG), electrocardiography (ECG), oxygen saturation in the blood, nasal airflow, and blood pressure.

EEG. Wafaa S. Almuhammadi et.al(Wafaa S.Almuhammadi, May 1, 2015) have done research on using EEG to classify OSA. They used EEG signals from Physionet as input for mining methods to detect if OSA exists or not. However, using only EEG for detecting OSA is not a good idea, because there are many health problems that have similar symptomatology to OSA, and one of them is epileptic (Karakis, August 29, 2012). EEG can be recorded by placing the electrodes along the scalp, which is uncomfortable compared to wearing smart devices. However, EEG is very useful in combination with the others observable signals. In other words, EEG is considered an important signal for detecting OSA.

EMG. People with OSA usually have longer time for the tongue to be recovered from relaxed. By examining the genioglossus muscle activity useful data for OSA diagnosis can be collected. Marc B Blumen et.al (Marc B Blumen, May 20, 2004) have found that those with OSA have tongue recovery time much longer than normal people after a constant submaximal effort, and they also have a smaller decrease in genioglossus muscle median frequency during effort. For OSA, we focus only on the signals from the tongue and genioglossus muscle because it causes the airway to be blocked. Other EMG signals are not considered, because they are not relevant.

EOG. To measure eyes movements during sleep electrooculography is used. Electrooculography has two electrodes. The front of the eyes, also known as the cornea, is electrically positive while the back, also known as the retina, is electrically negative. Therefore, it is possible to measure the variation in voltage to determine the movements of the eyes. Rapid eye movement (REM) is a factor in the occurrence of disordered breathing events(Babak Mokhlesi, 2012).

ECG. ECG presents the electrical activity of the heart, and it is one of the most efficient signal to detect OSA. By placing the electrodes on the skin, the tiny electrical changes on skin during each heartbeat can be detected. Laiali Almazaydeh et.al (Laiali Almazaydeh, May 2012) have done research on detecting OSA by using ECG signal features. They observe the wave forms from ECG, then identify an R peak. Once R peak is identified, they calculate RR intervals. After extracted, RR intervals are submitted to SVM in order to do classification.

Oxygen saturation. Oxygen saturation can be measured by using pulse oximetry. Pulse oximetry is a noninvasive and inexpensive procedure which is used to measure the level of oxygen in blood. It is very simple to make measurements, a clip-like sensor can be placed on the earlobe, nose or fingers(wikihow.com, August, 2016). The level

of oxygen in the blood is usually above 95%.

Nasal airflow. Airflow sensors can measure the air pressure in nose when inhaling and exhaling. The sensor can be easily placed, but it is uncomfortable to wear it during sleep. A lot of studies have reported that it is not a good signal for detecting OSA, but it is still used(Daniel de Sousa Michels, 2014). Therefore, we consider nasal airflow as one of signals to detect OSA.

Blood pressure. Blood pressure can be measured by a simple, noninvasive sensor. This sensor can easily clip on the finger. According to American Heart Association(heart.org, August, 2016), the normal blood pressure with systolic is less than 120 mm Hg, and with diastolic is less than 80 mm Hg. Getting higher number than these numbers when sleeping is considered having OSA. As explained in OSA causes, the pressure is high when the breathing pauses.

Chapter 3

Data sources and data formats

Signals for OSA come from many different sensor data sources. Moreover, the quality of these signals is diverse. Signals that come from clinical grade sensors are used in the medical domain, while signals that come from consumer electronic sensors are often used for fitness trackers, research, etc. In other words, consumer electronic sensors are not allowed in medicine, because they have not been approved yet. Therefore, a hypothesis is made that the signals from clinical sensors have better quality than the consumer electronic sensors. This chapter presents two corresponding sensor data sources for these two types of sensors. Consumer electronic signals come from BITalino sensors while clinic grade signals are stored in the Physionet database and captured with clinical grade sensors. In addition, different sensor data sources provide different formats to obtain and store the signals. Therefore, this chapter also presents a description of data formatting for each of the sensor data sources. In Section 3.1, an overview of data sources is presented. Section 3.2 presents different methods to format data signals. Standard formats, which are the European Data Format (EDF)(edfplus.info, n.d.[a]) and the European Data Format plus (EDF+) are presented in Section 3.2. Section 3.3 presents two data acquisition tools that can be used to obtain biosignals from sensor sources.

3.1 Data sources

In general, the quality of data has a huge influence to the data analysis process. Better analysis results are often derived from a higher quality data source. Therefore, choosing a good data source is vital for analyses. Misleading results are caused by various reasons, and a low quality data source is one of the major causes. However, a good source often comes with a very expensive price. In medicine, for obtaining biosignals has to follow strict requirements, and often use expensive clinical grade sensors to harvest these signals. The clinical data sensors provide high quality signals which can create a high quality dataset. After all, it is much more expensive to buy and maintain a clinical sensor system than a consumer electronic sensor system. For example, a NOX-T3 from nox-medical costs 55000kr plus value added tax(Thomas, August 26th 2016) while a BITalino costs about 1000kr. The consumer electronic sensors often come with reasonable or even low price, because the quality of signals

have not been approved, or maybe they are not good as the signals from the clinical sensors. However, there are many publications on using the consumer electronic sensors each year (for the BITalino 14 publications in 2015 and 7 publications in 2016(BITalino, 2016f)). Hence, an assumption is made that the quality of signals which is provided by the consumer electronic sensors is good enough to do research, build commercial applications etc. Nevertheless, the quality of biosignals and the support from consumer electronic sensors are promised to be improved in the future, because many leading companies constantly searching and integrated health care into their products, for example: Google Fit, Apple Health and Fitness, Microsoft band and Microsoft health, Xiaomi band, etc. This section presents an overview of BITalino which is one of the consumer electronic sensor groups for biosignals that has been used widely in many projects, products and researches, and Physionet database as clinical grade sensors source.

3.1.1 Data from BITalino sensors

BITalino is one of the reasonable priced sensor hardware platform which is easy to use and configure. BITalino is marked as DIY which means ”do it yourself”, therefore the goal of it is to use it for building an all-in-one platform such that every one can easily create their own biomedical devices. It is easy to configure, because BITalino is a plug-and-play device. The pre-conditioned analog outputs are sent wirelessly via bluetooth. Moreover, there are dozen of free, open source and paid softwares that support the BITalino sensor kit. These softwares can run on various operating systems, for example Linux, OSX, Windows, Android etc. Nevertheless, BITalino is not a medical device or to be used in medical diagnosis as they mentioned in their disclaimer on the website (BITalino, 2016d). It is for students, teachers, researchers, etc., who do not need to have any engineering electrical skills to use it.

BITalino Kit

BITalino has a wide rang of sensors. Those sensors have the capability to measure either bio-electrical or bio-mechanical signals. BITalino has three major components that together form a basic bio-harvesting device. The first component is the power management which provides and manages power for biosignals acquisition. The BITalino power component uses a 3.7V LiPo battery to power its analog and digital parts. The second component is the micro controller unit (MCU) which is designed for accurate and reliable real-time streaming. It can control up to 6 analog inputs, 1 output, 2 digital inputs and 2 digital outputs at up to 1kHz (BITalino, 2016e). The third component is data transfer. Data transfer is either bluetooth (BT) or bluetooth low energy (BLE), and is a ready-to-use module. The rest of the components are organized as independent modules that can be attached to the main board on demand.

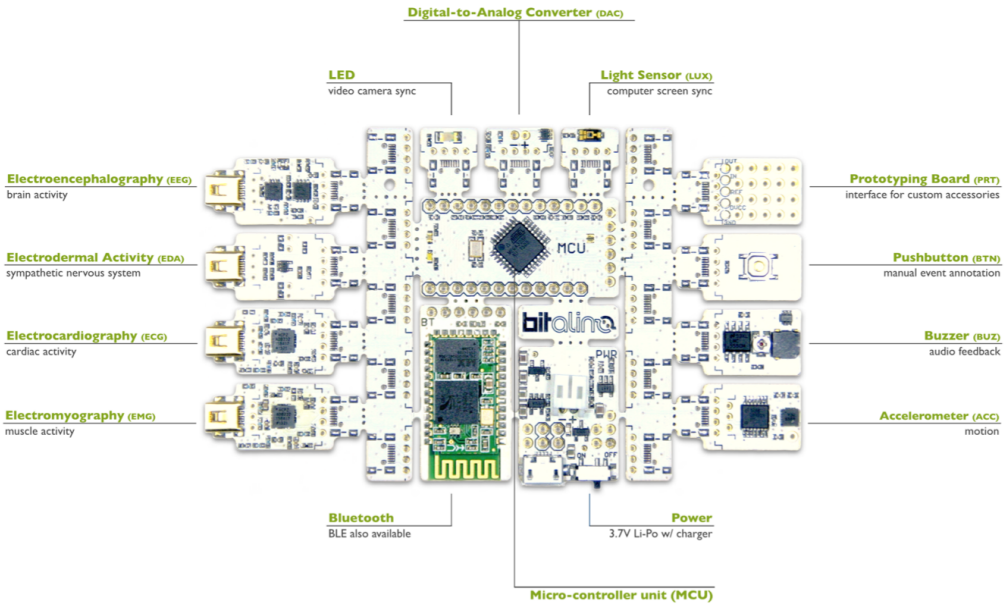


FIGURE 3.1: BITalino (r)evolution Board with Bluetooth connectivity (BITalino, 2016a)

BITalino provides three kits: BITalino (r)evolution board, BITalino (r)evolution freestyle and BITalino (r)evolution plugged.[1]

Figure 3.1 illustrates the BITalino (r)evolution board(BITalino, 2016a). The size of the board is 100x65x6mm, and it is powered by a 3.7V recharge battery. The board consists of analog ports (4in - 10bit, 2in - 6bit, 1in - battery, 1out - 8bit) and digital ports (2in - 1bit, 2out - 1bit). In addition, it has either BT or BLE which have a range about 10m. There are seven sensors which are integrated on the board. They are electromyography (EMG), electrocardiography (ECG), electro-dermal activity (EDA), electroencephalography (EEG), accelerometer (ACC), light (LUX) and pushbutton (BTN). The board kit is designed as all-in-one device. The board connects all the necessary components which are the power management, MCU, BT and sensors into one board which is ready to use. This kit is made for biosignal exploration and lab activities. In conjunction with the OpenSignals software, it provides real-time biosignal data visualization.

Figure 3.2 illustrates the BITalino (r)evolution plugged(BITalino, 2016c). It is similar to the board kit, however the main board contains only three major components which are power management, MCU and BT. The main board also provides analog and digital ports to the inputs and outputs. The sensors do not integrate on the main board, they are separated from the board and connected as plug and play on demand. Therefore, the plugged kit provides maximum flexible configuration.

[1]The evolution version is discontinued. The next generation of the BITalino is revolution, it has almost 2x the blocks and up to 60% smaller sensors.

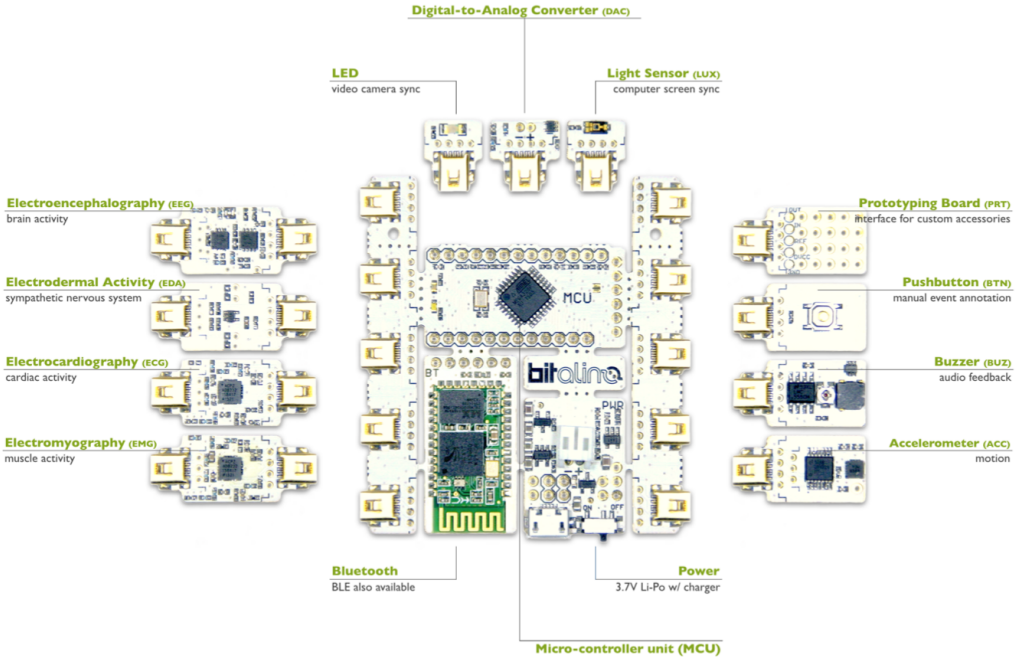


FIGURE 3.2: BITalino (r)evolution Plugged with Bluetooth connectivity (BITalino, 2016c)

Figure 3.3 illustrates the BITalino (r)evolution freestyle(BITalino, 2016b). As the name of the kit, it provides maximum configuration. Hence, the users can maximize their imagination in building wearable biosignal acquisition devices. Each component is separated in a individual module, even the power management, MCU and BT are not on the same board.

BITalino sensors

This section briefly describes the sensors, i.e., EMG, ECG, EDA, EEG, ACC, LUX, BTN, PZT, that are provided by BITalino.

The ACC sensor translates the motion into numerical values. BITalino provides 3-axis sensing for detecting tilt, monitoring activity, and measuring vibration. The sensor is limited in acquiring data from bio-mechanical and kinematic events. The sensor can be used in detecting posture, fall or shock, estimating rang of motion, step counting etc. Although it has 3 axis sensing, only the Z-axis is connected by default. It is because three analog outputs can be accessed separately, and the user can easily connect the X-axis and the Y-axis as their project demands.

The ECG sensor translates the bio-electrical signals, which are low amplitude and generated by a set of cells in the heart, into numerical values. The ECG sensor from BITalino can obtain data not only at the chest, but also at the hand palms. Furthermore, the sensor works with pre-gelled electrodes as good as most types of the dry onces. According to ECG sensor data sheet, the sensor provides bipolar differential

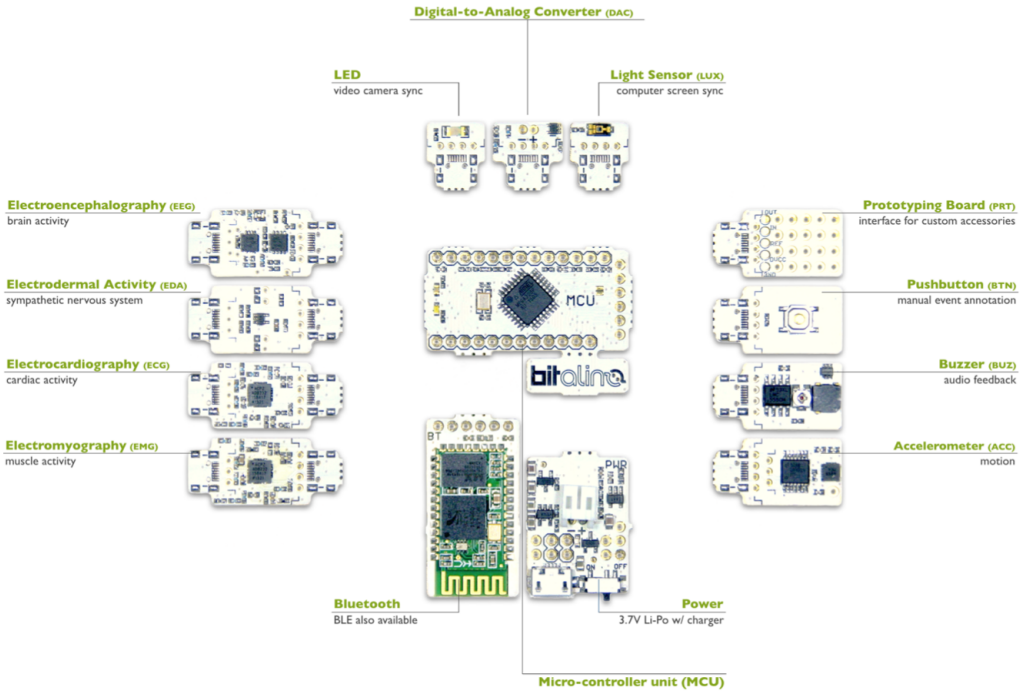


FIGURE 3.3: BITalino (r)evolution Freestyle with Bluetooth connectivity (BITalino, 2016b)

measurement with pre-conditioned analog output, and has a small form factor, and high signal-to-noise ratio. Therefore, it is widely used in heart rate and heart variability applications.

The EDA sensor translates the resistance of the skin into numerical values. The human body secretes sweat not only when the body needs to regulate temperature, but also when the sympathetic nervous system activity is affected, for example when relaxing or arousing. The BITalino EDA sensor provides pre-conditioned analog output with high signal-to-noise ratio from measuring skin resistance. This signal is widely used in many applications such as lie detector, relaxation etc. and in this thesis it is used for detecting arousal and emotional cartography.

The EMG sensor translates the bio-electrical signals which are sent from motor control neurons in the brain to the muscle fibers. A BITalino EMG sensor works with both dry and pre-gelled electrodes. This sensor has a wide range of applications such as muscle reflex studies, nerve conduction measurement, human-computer interaction etc.

The LUX sensor translates the intensity of light into a digital signal. The BITalino LUX sensor can be adapted to human eye responsiveness.

The push button is useful for taking the annotation of meaningful events which occur during the observation.

BITilano TMP sensor translates the temperature of a body or the environment into digital values which range from -40C to +125C. The accuracy of the sensor varies from -2C to +2C, and the linearity is 0.5. It is small (about 12x27mm) and has low

power consumption (about 0.05mA) which are good properties for building wearable devices.
PZT sensor translates the displacement variations induced when inhaling or exhaling into numerical values. BITalino PZT sensor consists of a adjustable chest strap, RJ22 connector and a sensor which is secured in the chest strap. A BITalino PZT sensor is a good option for respiratory analysis.

Logical sensors

A sensor network consists of dozens of sensor. Sometimes some of them need to be taken down for maintenance, or the network needs to integrate additional sensors to form a bigger network. A methodology has been introduced by C.Hansen et.al(C.Hansen, 1984) in November 1983 to deal with the problem, which is a logical sensor. The overall goal of logical sensors is to aid in the coherent synthesis of efficient and reliable sensor systems. There are two problems regarding to sensor systems(Tom Henderson, 1984). The first problem is how to build an efficient and coherent method for the information provided by various kinds of sensors. The second problem is how to maintain and develop the system for that it can be incorporated with additional sensing devices. Sensor data abstraction is introduced for solving these problems. Data abstraction techniques are used to analyze and create patterns (logical sensors) from sensor data(Payam Barnaghi, n.d.). The patterns are considered the interfaces for sensor systems which make the systems easily to be reconfigured in the future. Furthermore, the patterns together with the sensor semantic descriptions help to minimize the size of the data which is sent from the sensor nodes to the gateways or high level processing components. There are many principle motivations for logical sensor specification. Three of them are emergence of multi-sensor systems (a coherent data acquisition and integration system is needed), benefits of data abstraction (an inherent hierarchical structuring of logical sensors further aids system is needed) and availability of smart sensors (substitution of hardware for software and vice versa)(Carlos Carreiras, 2011).

3.1.2 Physionet sensor databases

As mentioned earlier, the quality of data plays a very important role in choosing data sets. Especially for biosignals, the data must be from the trustworthy sources. Therefore, Physionet sensor databases are chosen as trustworthy clinical grade sources. Physionet is inaugurated by the researchers at Boston's Beth Israel Deaconess Medical Center, Boston University, McGrill University and MIT(George B. Moody, n.d.). To date it is supported by the National Institute of General Medical Sciences and the National Institute of Biomedical Imaging and Bioengineering. Physionet has three interdependent components which are PhysioNet, PhysioBank and PhysioToolkit. The first component is PhysioNet which is used for exchanging and disseminating the biomedical signals and the software used to analyze those signals. PhysioNet also

offers training opportunities(PhysioNet, n.d.[b]) that consist of tutorials on a variety of topics, special designed data sets for classroom activities and annual open challenges. PhysioBank is the second component which contains biosignal databases from various kind of diseases. PhysioBank unceasingly grows in both size and scope, to date even signals from vivo and vitro experiments are accepted (PhysioNet, n.d.[a]). The last component is PhysioToolkit which provides tools for analyzing the biomedical signals. There are four databases in Physionet that can be used for analyzing OSA. They are apnea-ECG database, St. Vincent's University Hospital database, MIT-BIH polysomnographic database and SHHS polysomnography database.

Apnea-ECG database(T_Penzel, 2017):

This database has been created for Computers in Cardiology Challenge 2000 (CinC Challenge 2000). It was provided by Dr. Thomas Penzel of Phillipps-University, Marburg, Germany(Physionet, n.d.[e]). The data consist of 70 records that are equally divided into two data sets. One is a learning set, the other is a test set. Each record has a 100 Hz with 12-bit resolution ECG signal which is digitalized and lasts for slightly less than 7 hours to nearly 10 hours. In addition, the record also contains a set of apnea annotations. The apnea annotations are marked by physicians whether an apnea event happens or not, one annotation per minute. However, the apnea annotations are not available in the test set due to the challenge, but are made available after the contest. The annotations include age, gender, height, weight, apnea index, hypopnea index and apnea-hypopnea index. Eight records which are called a01 to a04, b01 and c01 to c06 have extra four additional signals(Physionet, n.d.[b]) for chest and abdominal respiratory effort, oronasal airflow, oxygen saturation in which the oxygen saturation signal digitized at 1Hz while the rest digitalized at 20Hz. Those additional signals are used as learning material to study the relationships between the respiration and ECG signals.

These records are harvested between 1993 and 1999 (T_Penzel, 2017). During 1998 to 1999, the ECGs were digitized at 200Hz. However, to synchronize with the ECGs which are taken during 1993 and 1995, the newer ECGs were decimated to 100Hz. Therefore all records in the data set have 100Hz ECG signal. The signals are from three groups of subjects: the apnea group, the "borderline apnea" group and the normal group. The mean age of these three groups is slightly different, the higher the mean age, the more severe the apnea. Furthermore, due to lack of episodes of pure central apnea or of Cheyne-Stokes respiration, obstructive and mixed apnea can not be distinguished in this data set, this is to say a hypopnea minute is the same as an apnea minute. Nevertheless, the Apnea-ECG database is a good source to use for studying and doing research on sleep apnea.

St. Vincent's University Hospital database:

St. Vincent's University Hospital Sleep Disorders Clinic has collected 25 full overnight polysomnograms records with three-channel Holter ECG. The clinical and demographic information were at first collected and assembled under the guidance of professor Walter McNicholas, Dr. Liam Doherty, Dr. Silke Ryan and Dr. John Garvey,

then scored and annotated by Ms Patrica Boyle, finally anonymized and electronic ally archived by Eric Chua. Signals which are monitored and stored are EEG, left EOG, right EOG, submental EMG, ECG, oronasal airflow, ribcage movements, abdomen movements, oxygen saturation, snoring and body position. The EDF format is used to save the records. Monitored subjects are above 18 years old, and randomly selected from patients referred to the clinic for that no known diseases can interfere with the heart rate. As a result, there are 25 subjects selected, i.e., 21 males and four females with the age range from 28 to 68 years, and the measured AHI range from 1.7 to 90.9(Physionet, n.d.[c]).

The MIT-BIH polysomnographic database:
The MIT-BIH polysomnographic database contains 16 subjects with sleep apnea syndrome from 60 male subjects in the age range of 32 to 56 years and a weight range from 89kg to 152kg(Y.Ichimaru, n.d.). This database is made for researchers who want to investigate clinical physiology, for engineers who want to develop a new method to analyze the digitized polysomnography data, and for students who want to learn sleep physiology. These subjects were observed in Boston's Beth Israel Hospital Sleep Laboratory for evaluating and testing CPAP. The monitored signals under observation are ECG, BP, EEG, Resp, EOG, EMG, SV and SO2 which have a sampling rate of 250Hz with 12-bit resolution. The recording time for each subject lasted from two to seven hours. This database is one of the most trustworthy resources for investigating and learning.

The SHHS polysomnography database: The Sleep Heart Health Study polysomnography database is used for determining the relationship between cardiovascular diseases and other consequences of sleep-disordered breathing. From 1995 to 1998, 6441 men and women with the mean age of 40 took part in the sleep-related breathing examination(Sleepdata.org, n.d.). The participants can stay at their home under extermination, and are monitored by trained technicians. The recoded signals include EEGs at 125Hz, EOGs at 50Hz, EMG at 125Hz, thoracic and abdominal movements at 10Hz, nasal-oral airflow at 10Hz, finger-tip pulse oximetry at 1Hz, ECG at 125Hz, heart rate at 1Hz, body position and ambient light. There is only one polysomnogram sample which is formated in the EDF form available in Physionet. However, more data can be downloaded freely via the National Sleep Research Resource website(sleepdata.org, n.d.). Since this database was made by a multi-center cohort study which are supported by the National Heart Lung & Blood Institute, it is also considered one of the most trustworthy resources for investigating and learning.

3.2 Data formats

When collecting signals, the selection of a structure for storing data is very important issue. The selected structure must provide fast accessing, support cross platform, extensibility, multi modality, querying etc. In other words, this selection depends on

the experiences of expert groups in implementing projects, the quality and the performance of the final result. Therefore, each group has its own data format for its special tasks. After choosing the data format, a set of rules are provided such that the database can be used outside the group. The problems raise when the number of groups increases. Hence, a standardized data format for biosignals is very essential for that these bio-research groups and clinics can not only coordinate and share data in an efficient way, but they can also improve the quality of the software.

3.2.1 Physionet sensor database formats

Although the stored data in the previously described four databases are biosignals, the format for each database is different.

Apnea-ECG database:

Apnea annotations are stored in the .apn and .qrs files(Physionet, n.d.[a]). In the .apn files, a one-minute interval is used between annotations. A "A" annotation presents that apnea was in progress at the beginning of the fellowed minute while a "N" annotation shows that apnea was not in progress at the beginning of the fellowed minute. Rdann[2] needs to be used for converting the binary annotations in these files into text for viewing. However, the .qrs files can be viewed by using sqrs125[3]. In the .qrs files, all observed heart beats are marked with "N" annotations, and QRS-like artifacts are marked with "|" annotations(Physionet, n.d.[a]).

St. Vincent's University Hospital database:

Unlike the way Apnea-ECG database stores its annotations (encoded its annotations), St. Vincent's University Hospital database has the annotations in a text file, the *_stage.txt and *_respevt.txt files. Figure 3.4 presents two tables for annotations in these files:

MIT-BIH polysomnographic database:

Each record in this database has two annotations files, there are .ecg and .st files. The .ecg file contains the beat annotations in wave form while the .st file stores the sleep stage and apnea annotations. Therefore, the .ecg files can be read by following WFDB Programmer's Guide which is made by George B.Moody(George, n.d.). The .st files contain note-annotations which are sleep staging and apnea information.

SHHS polysomnography database: The annotations of this database can be read by using PhysioBank ATM(physionet.org, n.d.) which is free to use from Physionet. The PhysioBank ATM can export the annotations to many different data formats, for example CVS, EDF, Matlab etc. The .hypn contains annotations for sleep stage, the .arou files contains the information on arousal event type and duration, the .oart files contains annotations which present oxygen saturation (SaO2) and their duration, the .resp files contains the annotations for apnea event together with

[2]Rdann is a program that reads and converts the binary annotations files into text, one annotation per line.

[3]sqrs125 is a program that locates QRS complexes in an ECG signal in the specified record.

Annotation	Meaning
0	Wake
1	REM
2	Stage 1
3	Stage 2
4	Stage 3
5	Stage 4
6	Artifact
7	Indeterminate

(a) Annotations for *_stage.txt(Physionet, n.d.[d])

1st column	Time of occurrence (time of day)	
2nd column	HYP	Hypopnea
	C	Central
	O	Obstructive
	M	Mixed
3rd column	Periodic breathing (PB)/ Cheynes-Stokes (CS)	
9th & 10th columns	Bradycardia/ Tachycardia	

(b) Annotations for *_respevt.txt(Physionet, n.d.[d])

FIGURE 3.4: Annotations for St. Vincent's University Hospital database

their duration and percent decrease in SaO2, minimum SaO2. The database has also .comp files that are compressed files. The .comp files contains all of the annotations that are stored in the .hypn, .arou, .resp, and .oart files.

3.2.2 EDF and EDF+ formats

In recent decades, many efforts to standardize the data format for biosignals have been done. One of them is the European Data Format (EDF). EDF is used for exchanging and storing multichannel biological and physical signals in a simple and flexible way. It was first introduced in 1987 at the Sleep Congress in Copenhagen and then contributed by all participating labs in August 1990(edfplus.info, n.d.[a]). The first publication of EDF was in 1992 in Electroencephalography and Clinical Neurophysiology 82, pages 391-393. The next version of EDF is EDF plus. It was published in 2003 and is compatible to the original EDF version. This is to say all existing EDF viewer programs can use EDF+. On the other hand, EDF+ can contain interrupted recordings as well as annotations, stimuli, and events. It has also fixed some problems that existed in EDF such as Y2K problem, comma vs dot, little-endian integers. Moreover, the EDF+ also supports UTF-8 format. The EDF+ is used in polysomnography, EEG, ECG, EMG and sleep scoring applications. Last but not least, EDF and EDF+ are free to use, and the users and developers can get supports from the EDF site via free downloads of files and software.
Both EDF and EDF+ have the same structure of a data file. As Figure 3.5 presents, the data file consists of two parts which are the header record and the data records. The first 256 bytes of the header part contains the basic information for the record which are the version of the data format (8 ascii), the identifications of the local patient and the local recording (80+80 ascii), start date formated as dd.mm.yy (8 ascii), start time formated as hh.mm.ss (8 ascii), the size of the header record (8 ascii), reserved field (44 ascii), the number of data records followed by the header record (8 ascii), duration of a data record (8 ascii), and the number of signals in data

```
HEADER RECORD (we suggest to also adopt the 12 simple additional EDF+ specs)
8 ascii : version of this data format (0)
80 ascii : local patient identification (mind item 3 of the additional EDF+ specs)
80 ascii : local recording identification (mind item 4 of the additional EDF+ specs)
8 ascii : startdate of recording (dd.mm.yy) (mind item 2 of the additional EDF+ specs)
8 ascii : starttime of recording (hh.mm.ss)
8 ascii : number of bytes in header record
44 ascii : reserved
8 ascii : number of data records (-1 if unknown, obey item 10 of the additional EDF+ specs)
8 ascii : duration of a data record, in seconds
4 ascii : number of signals (ns) in data record
ns * 16 ascii : ns * label (e.g. EEG Fpz-Cz or Body temp) (mind item 9 of the additional EDF+ specs)
ns * 80 ascii : ns * transducer type (e.g. AgAgCl electrode)
ns * 8 ascii : ns * physical dimension (e.g. uV or degreeC)
ns * 8 ascii : ns * physical minimum (e.g. -500 or 34)
ns * 8 ascii : ns * physical maximum (e.g. 500 or 40)
ns * 8 ascii : ns * digital minimum (e.g. -2048)
ns * 8 ascii : ns * digital maximum (e.g. 2047)
ns * 80 ascii : ns * prefiltering (e.g. HP:0.1Hz LP:75Hz)
ns * 8 ascii : ns * nr of samples in each data record
ns * 32 ascii : ns * reserved

DATA RECORD
nr of samples[1] * integer : first signal in the data record
nr of samples[2] * integer : second signal
..
..
nr of samples[ns] * integer : last signal
```

FIGURE 3.5: EDF and EDF+ file structure (edfplus.info, n.d.[b])

record (4 ascii). The rest of the header record (after the first 256 bytes) contains the information for each field of the signals. Each signal has a label which is not longer than 16 bytes, and coded in ascii. After the label is the transducer type which has the length up to 80 bytes. After the transducer type field is the physical dimension with its maximum and minimum observable values (both physical and digital), each of them is coded in 8 ascii bytes. The following 80 bytes are used to store filters which were used for the signal. The next 8 bytes are used for the number of samples in each data record. The last 32 bytes are reserved.

The data record part consists of a list of data records. Each data record has a list of samples for each signal. For each signal in the record, a sample for the signal is an integer. Hence, the number of bytes for each signal is the number of sample multiply with the size of integer.

The differences between EDF and EDF+ are the information in the 44 bytes reserved field and additional specifications are added to EDF+. In the reserved field, when a record is EDF+ and continuous, it must start with EDF+C, and EDF+D if it is discrete. There are 12 additional specifications in EDF+ which are some rules for the header, date-time, local patient and local recording identification, digital maximum and minimum, separator for digital grouping, endian format, standard texts and polarity rules etc.(edfplus.info, n.d.[c]).

The EDF+ data files are coded in a way that they can store both annotations and events. The EDF software can read EDF+ annotations as the physical sensor signals. Because the annotations are treated as signal, the results that are presented in the

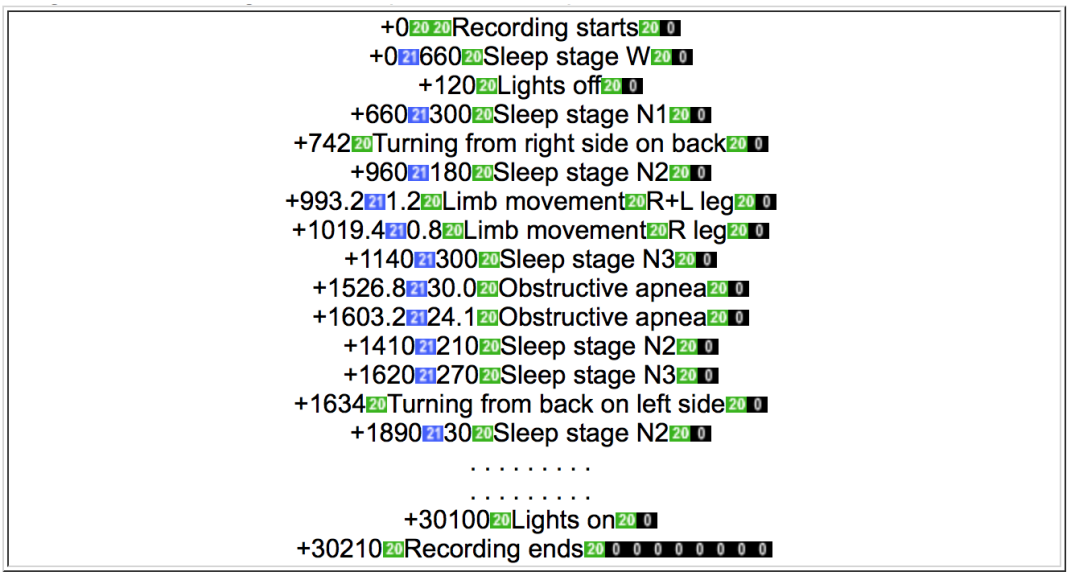


FIGURE 3.6: Sleep scoring sample(edfplus.info, n.d.[c])

EDF viewers are strange.

Beside containing ordinary signals, EDF+ can contain annotations signals. The annotations signal is defined by giving it the label "EDF Annotations". Then instead of storing ordinary signal samples, this field can be used for storing notation. This EDF annotations signal can be used for coding the text, time-keeping, events and stimuli as text. The annotations are kept in lists that are named Time-stamped Annotations Lists (TALs).

Figure 3.6 presents a sample of TALs for sleep scoring. The annotation signal is defined by giving it the label "EDF Annotations" in the label field in the header part. Then instead of storing ordinary signal samples, this field can be used for storing notation. This EDF annotations signal can be used for coding the text, time-keeping, events and stimuli as text. As presented in Figure 3.6, the annotations for sleep scoring are stored in TALs. Each TAL is formated in form +/-Onset21Duration20Annotation20Annotation20... Annotation20**0**, where 20, 21 and **0** are character codes. The Onset presents the amount of time before or after the starting time followed by a value 21 which is unprintable ASCII characters for ending the Onset. After the value 21 is the duration of this annotation. Both Onset and Duration can contains a dot character(float number) to make a better accuracy measurement. After the duration, each annotation can be separated by the unprintable ASCII character 20. The last annotation is followed by unprintable ASCII character 0 for terminating the list.

In EDF+, the data records do not need to be contiguous. Therefore, the start time of each record must be specific. Although the first annotation of the first EDF annotations signal is empty, the timestamp of the annotation must be specified how many seconds after the recording started in term of date and time. Hence, the first TAL in the first data record must start with +0.X*2020*, X can be dropped if it is 0.

3.3 CESAR data acquisition tools

First and foremost the sensor data which are generated by sensors need to be stored. Metadata needs to be defined such that the data can be easily sent, stored, and processed. This section presents two different tools that are used to obtain and store biosignals from different sensor sources (developed by Gjøby(Gjørby, 2016)), and from BITalino kit (developed by Carlos Carreiras et.al (Charles Hansen, 2011)).

CESAR is a project that is under developed by a group of universities (University of Oslo, National University of Singapore and Oslo University Hospital). In the Master Thesis by Gjøby(Gjørby, 2016), a generic data acquisition tool for mobile platforms (the Android platform) is presented. This tool is designed such that it is independent from data management and data analysis, and new data sources are also supported by it. An other tool which can be used to obtain biosignals for CESAR is StorageBIT. This tool was developed by Carlos Carreiras et.al (Charles Hansen, 2011). StorageBIT is not only a data acquisition tool, but also a database model used for storing biosignals in an efficient way. Regarding the scope of this section, only data acquisition part of StorageBIT is presented in the section.

3.3.1 Generic Data Acquisition for Mobile Platforms

Gjøby finished his master degree with the project title ”Generic Data Acquisition for Mobile Platforms” at the Department of Informatics at the University of Oslo in spring 2016. He worked on designing and developing an extensible system which allows applications to collect external and built-in sensor data through one common interface. He chose to design and implement a system for Android platform in his master thesis. He presents the overview of the system in the chapter ”Not Android Specific”. In this chapter, he explains how to design sensor wrappers and providers. A sensor wrapper establishes a connection to a specific data source in order to collect data, after that it sends the collected data to the provider application. Hence, the sensor wrapper needs to be designed to fit the technology of the link layer and communication protocol of the specific data source. Therefore, each data source has its own sensor wrapper. A provider is an extended sensor wrapper management which can use any of the available sensor wrappers. Figure 3.7 illustrates an overview of how sensor wrappers connect to a provider, and a way provider serves multiple applications.
In the chapter ”Android Specific”, he implements this design on the Android platform after discussing on the suitable components and functions that Android provides. To collect data from physical sensors and send the collected data to a user application, he defines two JSON structures as illustrates in the Figure 3.8.
His application immediately sends a metadata to the application when it receives a connection demand. He defines the metadata in form of a JSON-Object which has as the first element a tuple(”type”,”data”). The second and third elements are name

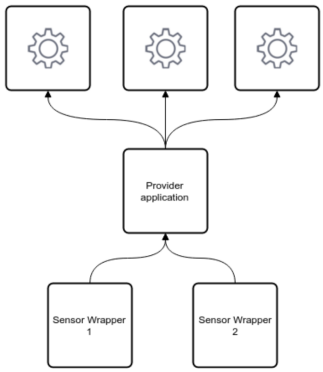


FIGURE 3.7: Sharing the collected data between multiple applications(Gjørby, 2016)

and ID of the sensor wrapper, the remaining elements are the data from the channels, which belong to the sensor wrapper, in form of a JSON-Object array. Each channel has a unique id, type (ACC,ECG,EMG, etc.), metric and description. The JSON-Object is converted to a string and sent to the application. After connected to the BITalino board, the application retrieves data from physical sensors and submits to the application. The connection to the BITalino board should be stopped after receiving a stop signal from the application.

As Gjøby presents in chapter "Not Android specific", the context of the data acquisition is specified as metadata for that it can be used for one common interface. After a comparison between JSON, XML and binary buffer, he decided to use JSON for streaming sensor data in his tool. At first, a metadata which contains general, unchanged information under acquisition (also known as a described context of the collected data from sensor wrapper) is sent, then the collected data from sensors are sent as soon as they are available to avoid overhead. The tool provides a common interface by combining the sensor wrapper ID and the channel ID as abstract level to identify each channel in the system. Figure 3.9 presents a sample of the metadata for BITalino by using a JSON structure. An existing sensor database can be treated as a sensor wrapper with a unique ID. The requested data for a channel in the sensor database can be obtained by using the database ID together with the ID of the requested channel. In other words, the metadata in form of JSON structure for all sensor sources are the same such that they can be used by a provider via one common interface.

Each data sample contains type, sensor wrapper id, timestamp, and the data for the channels. The type of the sample must be "data" such that it can be distinguished with the metadata package which has the type "meta".

Another important component in this tool is a provider. The provider can be considered as a bridge to connect the sensor wrappers with the applications. The provider works as a sensor wrapper management and data processing. It discovers sensor wrappers dynamically by broadcasting a discovery message, then it registers the responded sensor wrappers for later harvesting data. The collected data from sensor wrappers

are pushed to the provider by using earlier mentioned JSON structure. Then, the provider processes the data, i.e, stores or forwards the data to an application.

3.3.2 StorageBIT data acquisition

Carlos Carreiras et.al (Charles Hansen, 2011) have done research on storing biosignals from BITalino. This tool can convert sensor data from different sources into their data model. Although the tool mainly supports the BITalino kit, it can also manage other sources by mirroring these sources into its data model. At first they make a comparison of the existing data formats which are the Extensible Biosignal File Format (EBS), the European Data Format (EDF and EDF+), the Medical Waveform Format Encoding Rules (MFER) and the Waveform Database(WFDB) based on eight properties which are compression, non-sequential access, cross platform support, events support, extensibility, metadata, multi modality and querying. After that, they make a data model which has ”Records” as the basic abstract entities for storage. Each record consists of header, audit, biosignals and events and is stored in form of JSON-Object as presented in Figure 3.10.
The header contains the basic information such as the patient identification, when and where the record was taken, and who performed the acquisition. The audit field contains the history of the file such as the applied filters or processing steps etc. The biosignal filed contains the data of the interested signals such as observing duration, sample rate, the label of the signal, transducer etc. The events field contains any asynchronized information like annotations or any events which are not synchronously from time series.

```
{
   "type": "meta",
   "name": "InPhoneSensor",
   "id": 0,
   "channels": [
         {
               "id": 0,
               "type": "ACC",
               "metric": "G",
               "description": "Phone, X"
         },
         {
               "id": 1,
               "type": "ACC",
               "metric": "G",
               "description": "Phone, Y"
         },
         {
               "id": 2,
               "type": "ACC",
                  "metric": "G",
               "description": "Phone, Z"
         }
   ]
}
```


(a) A JSON structure describing the encoding of the metadata


```
{
   "type":"data",
   "id":0,
   "time":"13:28:59:365"
   "data":[
         {
               "id": 0,
               "value": 0.057708740234375
         },
         {
               "id": 1,
               "value": -0.0457763671875
         },
         {
               "id": 2,
               "value": 9.865188598632812
         }
   ]
}
```


(b) A JSON structure describing the encoding of a data reading

FIGURE 3.8: A JSON structures used to send and receive sensor data.(Gjørby, 2016)

```
{
   "type": "meta",
   "name": "BITalino",
   "id": 2,
   "channels": [
               {"id": 0,
                "type": "ECG",
                "metric": "mV",
                "description": "NONE"},
               {"id": 1,
                "type": "TMP",
                "metric": "C",
                "description": "NONE"},

               ....

               {"id": 7,
                "type": "EMG",
                "metric": "mV",
                "description": "NONE"}
              ]
}
```

FIGURE 3.9: A JSON structure sample of the metadata from BITalino

```
Record = {
  'Header': {
    'Subject': <Patient ID>
    'Record ID': <Record ID>
    'Date': <Start Date> + <Start
        Time>
  },
  'Biosignal 1': {
    'Duration'
    'Sample Rate'
    'Label': <Label [1]>
    'Transducer': <Transducer [1]>
    'Physical Dimension': <Physical
        Dimension [1]>
    'Physical Maximum': <Physical
        Maximum [1]>
    'Physical Minimum': <Physical
        Minimum [1]>
    'Digital Maximum': <Digital
        Maximum [1]>
    'Digital Minimum': <Digital
        Minimum [1]>
    'Audit': <Prefiltering [1]>
  },

  ...

  'Biosignal NS': {
    'Duration'
    'Sample Rate'
    'Label': <Label [NS]>
    'Transducer': <Transducer [NS]>
    'Physical Dimension': <Physical
        Dimension [NS]>
    'Physical Maximum': <Physical
        Maximum [NS]>
    'Physical Minimum': <Physical
        Minimum [NS]>
    'Digital Maximum': <Digital
        Maximum [NS>
    'Digital Minimum': <Digital
        Minimum [NS]>
    'Audit': <Prefiltering [NS]>
  }
}
```

FIGURE 3.10: Mirroring of the EDF+ file structure onto the Data Model (Charles Hansen, 2011)

Chapter 4

Database modeling

It is well recognized that the design of a database has a huge influence on the quality of the database applications. To design a database means to build a formal model for the database application. The data model does not only define a logical structure of a database, but also determines a set of rules and operations which could be used for performing actions on the data. All of data items that have meaning in the real world can be stored in a file system. A file is a collection of the data items, and need to be managed in a way that can be easily read and updated. A relational database is one of the ways that are used for managing files. In the relational database, a smallest unit of data in the real world can be mapped into an attribute that belongs to a certain entity in a database. In term of database' components, the smallest unit of data is called a column, a group of related columns is called a tuple or a row. Each row reflects to a specific object in real world, and these rows are abstracted into a table. In other words, the table presents an entity type in the database system.
To make the analysis of OSA data easier, a database system is taken in this work into considering. The mentioned data sources in Chapter 3 use files to store their bio-signal data. As a result, they do not take advantages of the offered functions of a database management system for data analysis. On the other hands, each source can be used by a single user at the time, and therefore, to compare the quality of sources is very difficult. Hence, the need of storing the OSA data into tables in a database system must be seriously considered. This chapter presents a data model for storing OSA bio-signals by analyzing the requirements of users and the format of data sources. Based on the analysis, a data modeling procedure is performed to find the most suitable database model for storing OSA data.
Section 4.1 presents requirements for the OSA database, in which the requirements are grouped into group of users, and the requirements of the sensor sources. Section 4.2 presents conceptual data modeling. In this section, entities and their relationships are defined. From that, a logical model is derived as presented in Section 4.3. The logical model is independent from a database management system. It only presents the structure of the database, therefore database conversion and reorganization are much easier to be taken. In Section 4.4, a specific database management is chosen to implement the logical database model. Since an Android mobile platform is used to collect OSA samples (CESAR acquisition tool), and the thesis would like to build a

database application for CESAR on mobile platform, SQLite database management system is chosen to implement the database model.

4.1 OSA database system requirements

When designing a database, it is essential to identify requirements. The requirement for a database system usually focus on two things, that are what the database is to be used for, and what it must contain. In terms of storing OSA data, the data system must satisfy requirements of the sensor sources (what it must contain), and requirements of users who using it (what the database is to be used for). There are three main groups of user, which are patients, physicians, and researchers. These users would like to have a database system to keep track of collected bio-signals from the CESAR acquisition tool (BITalino sources) and other sources, e.g., in form of the EDF format (Physionet databases). The database system must support the future changes in OSA diagnostic such that the system does not need to be rewritten.
Since the source data for the database system are the CESAR acquisition tool and the EDF/EDF+ file format, the system must at least store all data items from them. The data items of the CESAR acquisition tool and the EDF/EDF+ file format are presented in Table 4.1, where derived data items in EDF format such as number of bytes in header, number of data records, etc., are not included in the table. The description column presents the interested objects, and these objects must be stored in the database system.

In terms of what the database is to be used for, the user requirements need to be considered such that the database can be designed in a way it satisfies the requirements of the users. The requirements can be derived from activities the users perform on data in the system. Table 4.2 presents a list of possible actions the patients, physicians, and researchers interact with the database system.
To make it is easy to follow, a brief description of the actions that are presented in Table 4.2 is taken into discussion. The mains activities of the patents on the database system are functions that do inserting, simple querying, and deleting. Therefore, most of their actions are import and export. Some of functions (pre-define queries) can be defined for the patients. Hence, the patients are not allowed to use self-defined functions. The physician, on the other hands, can execute modifying functions, because they are allowed to manually train the data. In other word, the physicians have knowledge on OSA health problems, and they know which signals are abnormal. Therefore, the system must support modifying functions such that the physicians can quickly update the abnormal signals. In contrast, the main focuses of the researchers neither inserting nor updating functions (they also use them, but they are not the main focuses). The researchers often perform evaluating tasks on the system to find the best solutions for future use. In term of OSA database system, they would like to evaluate the quality of sources that are used for collecting bio-signals. Low quality sources must not be further considered, since they provide inaccuracy data, which

Description	CESAR acquisition tool	EDF/EDF+ file format
The identifier of a source	Wrapper id	n/a
Name of source	Wrapper name	File name
The identifier of a channel	Channel ID	n/a
Name of channel	Data type	16 ascii: signal label
Metric	Metric	8 ascii: physical dimension
Recording description	Description	44 ascii: reserved field
Information of patient	n/a	80 ascii: local patient identification
Information of clinic	n/a	80 ascii: local recording identification
Recording fragment	n/a	Data record
Fragment duration	n/a	8 ascii: duration of a data record, in second
Number of used channels	Can derived from metadata package	4 ascii: number of signals in data record
Transducer type	Can derived from BITalino documents	80 ascii: transducer type
Physical maximum	Can derived from BITalino documents	8 ascii: physical maximum
Physical minimum	Can derived from BITalino documents	8 ascii: physical minimum
Digital maximum	n/a	8 ascii: digital maximum
Digital minimum	n/a	8 ascii: digital minimum
Pre-filtering	n/a	80 ascii: prefiltering
Number of samples in a fragment	n/a	8 ascii: number of samples in each data record
Other information for a channel	n/a	32 ascii: reserved
Timestamp for a sample	Can derived from data package	Can calculate from timestamp of the recording
Sample value	Float type value	2-byte integer, a guide to convert between these bytes to float and vice versa
Annotation onset	n/a	onset from Time-stamped Annotations Lists (TALs)
Annotation duration	n/a	duration from Time-stamped Annotations Lists (TALs)
Annotation timekeeping	n/a	Time keeping of data records
Annotations	n/a	Annotations in a TAL

Table 4.1: A summary of data items from CESAR acquisition and EDF/EDF+ file format

User group	Action
Patient	- Find recordings, physicians, clinics - Store samples from CESAR tool - Retrieve samples for certain channels - Import their previous recording - Export certain sources or channels - Export part of a recording for some channels - Delete a specific source - Delete them self from database - Import trained OSA data from physician - Execute mining to find out OSA for a new recording based on a trained source
Physician	- Find patients, recordings, clinics - Import EDF file from patient - Store samples from CESAR tool - Retrieve all recording for a patient - Retrieve samples for a channel from different patients to do comparison - Retrieve part of recordings for some channels - Update annotations for a recording - Export recoding to EDF file to share with other physician or researcher - Delete a source, a recording, a patient, etc. - Manual training data for a recording by taking annotations while visualizing sources - Execute mining algorithm to find AHI for new sources
Researcher	Beside the basic actions as the patients and physician have, researchers can perform more advance actions such as: - Evaluate the quality of sources and channels that are used for collecting sample - Perform raw query to find out the best query algorithms, which cost minimum resources when executed, minimum running time - Apply possible mining algorithms to find AHI for the database model which implemented in a specific database management system - Evaluate the possibility of database system when implemented on different hardware mobile platform - Develop new database applications based on the database model

Table 4.2: A summary of user requirements

lead to many problems when performing data analysis. The researchers can also evaluate different algorithms with respect to resources used, performance, etc. on the database system, since an algorithm can be good for a specific system, but behave badly on the other systems. An algorithm should not be chosen randomly; it must be evaluated carefully. Nevertheless, the database design can be used by different database applications on different operative system platforms.

4.2 Conceptual data modeling

"Conceptual modeling is about describing the semantics of software applications at a high level of abstraction"(Thalheim, 2011). Similarly, conceptual data modeling can be understood as the semantics of a database at a high level of abstraction. Therefore, this section describes only entity names and their relationships. Attributes and keys are not included in the section to maximize the abstraction. As described in Section 4.1, and as presented in Table 4.1 and 4.2, data items that the database system must keep track of are source identifier, source name, channel identifier, channel name, metric, recording description, patient, clinic, recording, recording timestamp, recording fragment, recording fragment duration, transducer type, physical maximum, physical minimum, pre-filtering, number sample in a fragment, reserved information for a channel, sample timestamp, sample of recording, physician who works for clinic, annotation, patient name, patient gender, patient date of birth, patient code in clinic (patient information from EDF), clinic code, physician code, and used equipment by clinic (clinic information from EDF). However, not all of the data items are considered as entities of the database system. As Toby et.al(Toby Teorey, 2006) presents in their Database Modeling and Design book, an entity should contain descriptive information while an attribute is not. An attribute is a data element which requires only one identifier, in addition, an attribute does not have any relationships. On the other hand, multivalued attributes are classified as entities, and attributes must be attached to the closest entities they describe. The Patient entity and Physician entity have a lot of common attributes, it is to say, they are Person. Therefore, a new entity, which is Person, must be defined such that the Patient and Physical can extend from this new entity. As a result, Person is added to the set of entity of the database system.

Based on their guidelines, the data items can be classified into entities and attributes as presented in Table 4.3. In terms of future analysis, some attributes are added to the entities as meta-data for later analysis such as source type, recording at a frequency, etc. Figure 4.1 presents an example of a recording, in which, Channel 1 and Channel 2 present ordinary signals while Channel 3 presents annotation for the recording. Based on timestamp of samples, a fragment of the recording can be derived. Therefore, metadata for a data record from EDF are not need to define. One of the most important part of the conceptual data modeling is to define relationships between the classified entities. In this part, the Object Role Modeling (ORM) is

Entity	Attribute
Source	source identifier, source name, channel number, channel name, metric, transducer type, physical maximum, physical minimum, digital maximum, digital minimum, EDF reserved compatible
Recording	recording id, source identifier, channel number, id person collects recording, id person own recording, recording timestamp, recording description, frequency, pre-filtering, sample timestamp, sample value, annotation onset, annotation duration, annotation timekeeping, annotation text, used equipment, EDF reserved for recording
Person	person id, name, city, phone number, email, gender, data of birth, age, height, weight, BMI, patient id, physician id, other health issues, title in clinic
Clinic	clinic code, name, address, phone number, email

TABLE 4.3: Classified entities with their attributes

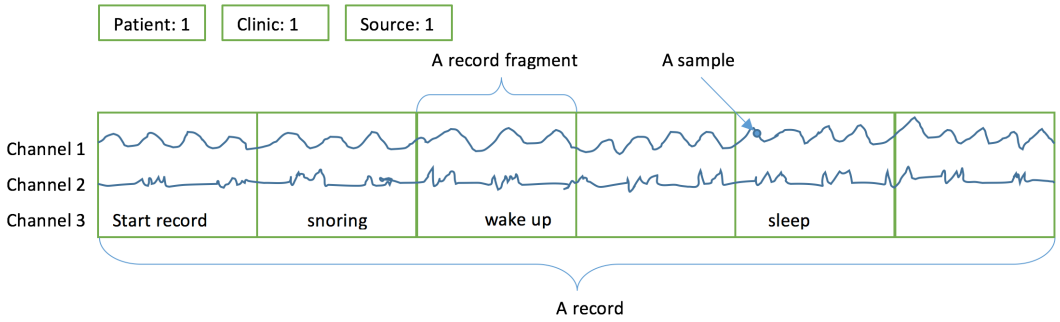


FIGURE 4.1: Example of a recording from a source

used for presenting the relationships between entities. ORM is chosen because it is a method for designing and querying database models at the conceptual level, and easy to validate and evolve(Halpin, 2017a).

A short history of ORM, the term "object-role model" is given in Eckhard Falkenberg's doctoral thesis which was published in 1976(Wikipedia, 2017a). ORM is a very good method which is used for designing and querying database models at the conceptual level. ORM uses natural language, as well as diagrams to simplify the design process. With ORM, a conceptual approach to modeling is provided by expressing the model in terms of natural concepts, such as objects and roles. Elementary facts are fundamental for ORM. These elementary facts are expressed in diagrams, and are verbalized into natural language. Nevertheless, modeling, transforming, and querying data from a domain becomes much easier with the help of the "fact-based" approach. ORM is easily to understand by non-technical users, because it is attribute free. It is to say, all the facts are treated as relationships. Moreover, when drawing a graphic, it is more expressive and easier to be understood by people without technical background. Last but not least, avoiding attributes in the database model does not only

improve the semantic stability, but also enables the verbalization into natural language.
Based on the classified entities, facts of the database system can be expressed as below:

- Source has Recording for Patient(Person) at Clinic.
- Recording for Patient(Person) is produced by Physician/Patient(Person) by using Source.
- Patient/Physician(Person) at Clinic uses Source to produce Recording.
- Physician(Person) work for Clinic.
- Patient(Person) belongs to Clinic.

There is another view of the conceptual data model where the entity Recording is treated as a relationship. Facts that respected to this view are as below:

- Source is used by Patient(Person) at Clinic.
- Patient(Person) at Clinic uses Source.
- Physician(Person) uses Source for Patient(Person).
- Physician(Person) work for Clinic.
- Patient(Person) belongs to Clinic.

The second view is easy to read. However, when transform this view into a logical model, a product from "uses" must be defined, and it is a Recording with respect to the first view. It is to say, there are a lot of possible views when doing conceptual data modeling. It is difficult to say which is the best view to use. Therefore, these views need to be integrated. Toby et.al[cite] also suggest four steps for conceptual schema integration, they are pre-integration analysis, comparison of schema, conformation of schema, merging and restructuring of schema. Integration for the views in this part is simple. The first view is extended from the second view, hence, it has a better presentation compared with the second view.
Figure 4.3 presents the first view where Recording is presented as an entity, while in Figure 4.4, Recording is presented as a relationship. As argued in view integration,

Figure 4.3 illustrates the conceptual model of the database system. Only notations that are used in the thesis, are described. The other notations which are irrelevant to this design, can be found in the ORM article written by Terry Halpin(Halpin, 2017b). Figure 4.2 presents the used notations, which are uniqueness constraint, object type shapes, shapes and readings for predicates and roles, sub-typing, and mandatory role constraint.

In the figure, **uniqueness constraint** (a) is used in the first version of ORM, while (b) is used in the second version(ORM2). These notations which are many to many (n:m), one to many (1:n), one to one (1:1), one to one which is primary, and a combination of these constraints respectively to the figure.
Object type shapes (a) is used in ORM, while (b), (c), and (d) is used in ORM2. As presented in ORM2 article(Halpin, 2017b), after a survey of 18 experts, 12 of them prefer (b), 5 of them prefer (d), and the last one prefer (c). Hence, (c) is chosen as default type shape for objects, while (c) and (d) are the alternatives.
Shapes and reading for predicates and roles (a) are used in ORM, and (b), (c) is used in ORM2. There is a bi-directional read for predicates and roles.
Mandatory role constrains are indicated by a solid dot. (a) is notations for ORM, and (b) is for ORM2.
Sub-typing notation presents the hierarchy between objects.

4.3 Logical data modeling

The logical data model describes the abstract structure of the database system. Therefore, it is independent of a particular database management system. That means it does not describe what data types should be used, which technologies can be used such that queries can execute fast, etc., but it should describe tables (entities) and columns (attributes), relationships, etc., in which the primary keys for columns and the reference keys must be specified. The logical data model also the presents relationships between the entities. Therefore, all entity relationships need to be specified. Then, all attributes for each entity must be carefully identified. Since this thesis takes the future growth of data and meta-data of OSA database into considering, it is essential for finding all possible attributes for the specified entities that are described in Section 2. After that, a set of function dependences (FDs) can be derived from the relationships and attributes. Finally, database normalization need to be performed such that the database can be accurate, fast, and efficient. Subsection 3.1 presents the relationships between classified entities. In this subsection, many-to-many relationships are also resolved. Normalization and step by step to normalize are discussed and presented in Subsection 3.2.

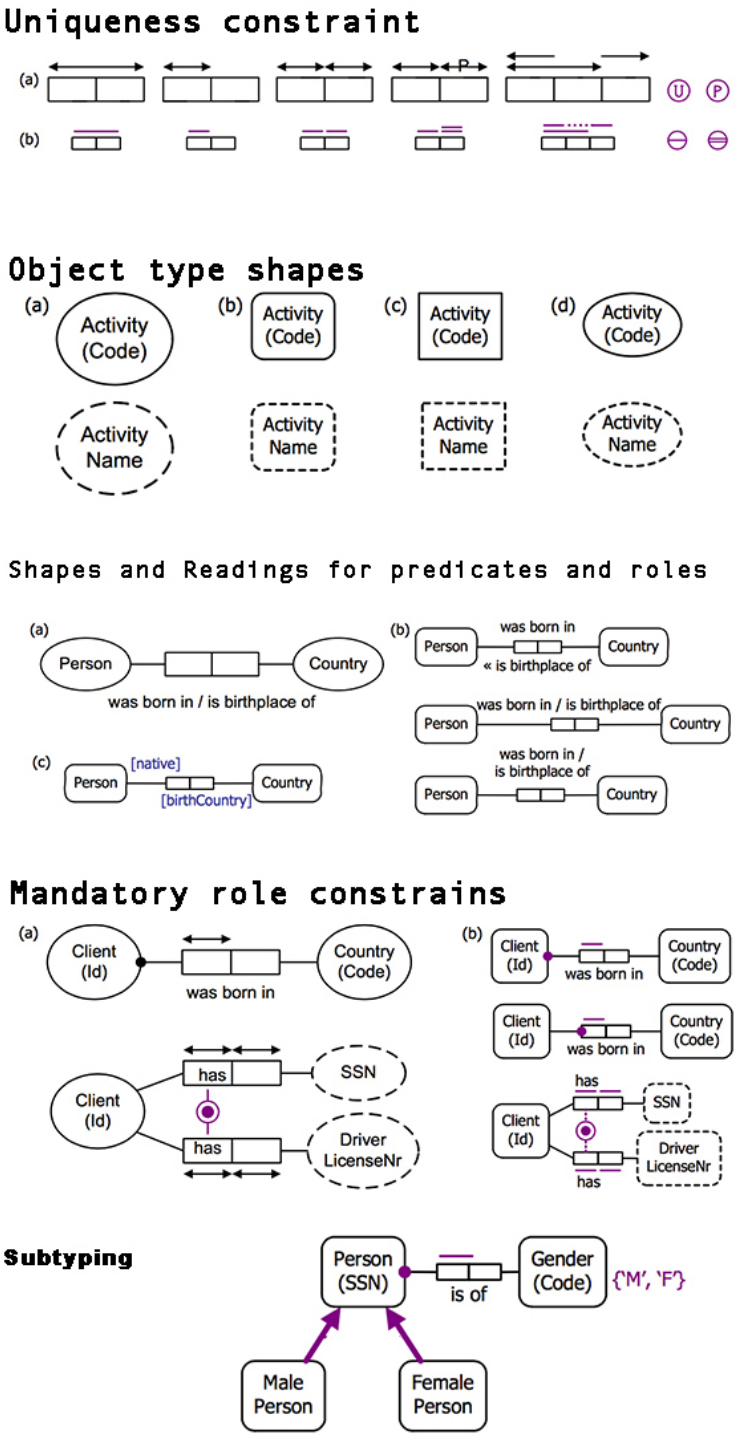


FIGURE 4.2: View where Recording is presented as an entity

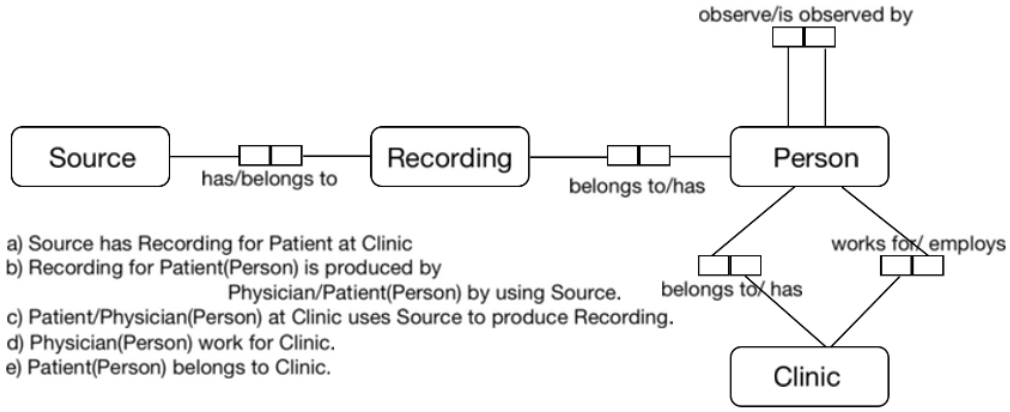


FIGURE 4.3: View where Recording is presented as an entity

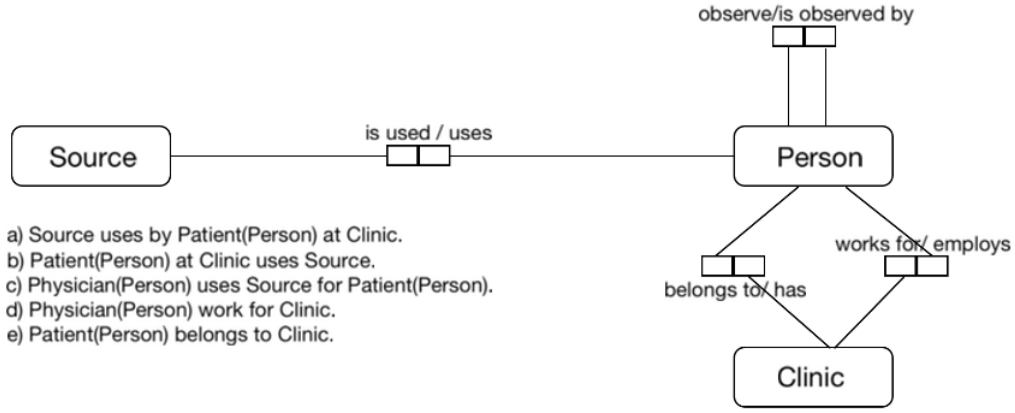


FIGURE 4.4: View where Recording is presented as a relationship

4.3.1 Relationships between different entities

Based on the facts that are presented in Section 2, the data relationships are described as the following assertions:

- Each Source has many Recordings, but one Recording belongs to only one Source.

- Each Source can be used by many different Person, and each Person can use many different Sources.

- Each Source can be used by many different Clinics, and each Clinic can use many difference Sources.

- Person has many Recordings, but one Recording belongs to only one Person.

- Person collects many Recording, but one Recording is collected by only one Person.

- Each Recording is produced by a Source, for a Person at a certain Clinic at a certain time.

- Each Person works/belongs to many Clinics, and each Clinic employs/has many Person.

- Each Person (Physician) observers many Persons (Patient), and many Person(Patient) are observed by a Person(Physician).

These relationships can be divided into three groups: binary relationships, binary recursive relationships and ternary (or n-ary) relationships. Figure 4.5 and 4.6 present these relationships by using ORM notations.

Foreign keys are easily derived from binary relationships. If there is a one-to-one

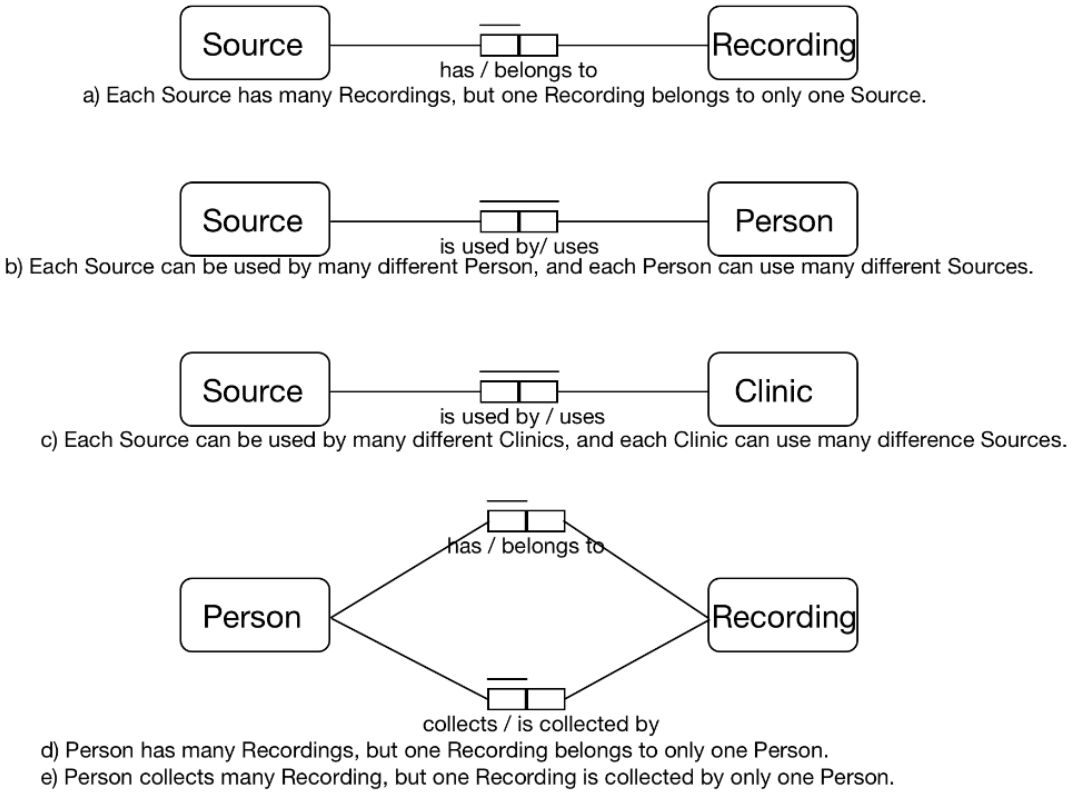
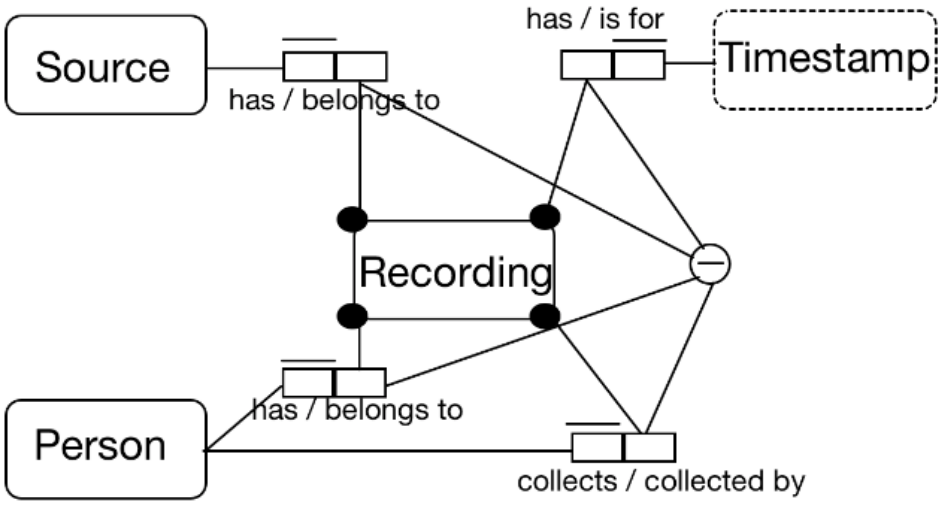
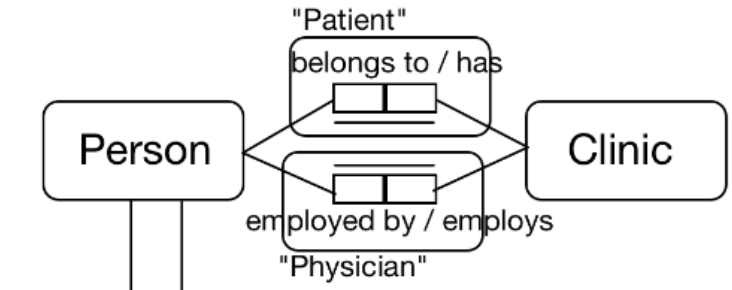


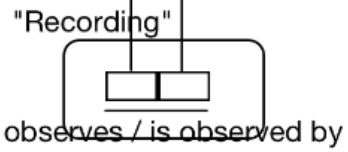
FIGURE 4.5: Binary relationships

binary relationship, the key of either entity can be used as a foreign key in the table of the other entity. If there is a one-to-many binary relationship, the foreign key must appear on the "many" side, since the "many" side presents the child entity.



f) Each Recording is produced by a Source, for a Person at a certain Clinic at a certain time.





g) Each Person works/belongs to many Clinics, and each Clinic employs/has many Person.
h) Each Person (Physician) observers many Persons (Patient), and many Person(Patient) are observed by a Person

FIGURE 4.6: Recursive and n-ary relationships

A many-to-many binary relationship must be resolved, since the relational database management system cannot hold this relationship. An easiest way to resolve a many-to-many binary relationship is to convert this relationship into a new entity, in which each old entity has a one-to-many binary relationship with the new entity. In this thesis, it is natural to see that most of the many-to-many binary relationships from the assertions can be resolved by using the Recording entity as a new entity between the old entities. Physician and Patient can be used as new entities for relationships "works/belongs to" and "employ/has" respectively in the assertion "Each Person works/belongs to many Clinics, and each Clinic employs/has many Person". As a result, the new entities must contain foreign keys that refer to parent entities.
In a binary recursive relationship between two Persons (a Person observes a Person), a foreign key refers to a column which identifies the referred Person. The 4-ary relationship "Each Recording is produced by a Source, for a Person at a certain Clinic at a certain time" presents that Recording is depended on a Source, a Person and a Clinic at a specific time, therefore it must contain the primary keys of these entities as foreign keys.
A multivalued dependence A $\twoheadrightarrow$ B is defined whenever a relation has two tuples that agree in all the attributes of X, then their Y components can be swapped and get to new tuples that also in the relation. In entity Recording, a recording id $\twoheadrightarrow$ all of the annotations attributes. That is, there are possible to have multiple annotations at the same timestamp, and many timestamp could have the same notation text. Therefore, entity Recording has one multivalued dependence which is recording id $\twoheadrightarrow$ annotation onset, annotation duration, annotation timekeeping, annotation text. Most of attributes for each entity can be taken from Table 4.3. In terms of future analysis, some of attribute are added to Person. Table 4.4 presents the entities together with their attributes and FDs, and it is a result of the analysis. In this table, each attribute of a entity is given an alias for saving writing when finding primary keys and decomposition.

Based on the defined function dependents of the entities, primary keys of these entities can be found by following these steps(Munthe-Kaas, 2017):

With FDs belong to a relation/entity R:

1. Let X = a set of attributes that are not exist in any right hand side if the FDs.

2. Expand systematic X in every possible ways with the attributes that occur at least one on left hand side of FDs.

Compute closure X+ for each such X until X+ are all attributes.

If X+ are all attributes in R, check that whatever an attribute A is chosen in X, (X-A)+ is not a set of all attributes of R.

If that is the case, X is a candidate (primary) key.

A algorithm, that is used for computing a closure of the attribute set with respect to FDs, is as follow(Munthe-Kaas, 2017):

Let X is a set of attributes in relation, and X+ is a closure of X.

Entities	Attributes	Function dependents
Source	source identifier (A), source name (B), channel number (C), channel name (D), metric (E), transducer type (F), physical maximum (G), physical minimum (H), digital maximum (I), digital minimum (J), EDF reserved compatible (K), source type (L)	**A,C** → D, E, F, G, H, I, J, K **A** → B, L
Recording	recording id (A), source identifier (B), channel number (C), id person collects recording (D), id person own recording (E), recording timestamp (F), recording description (G), frequency (H), pre-filtering (I), sample timestamp (J), sample value (K), annotation onset (L), annotation duration (M), annotation timekeeping (N), annotation text (O), used equipment (P), EDF reserved (Q)	**A** → B, C, D, E, F, G, H, I, L, M, N, O, P, Q **B,C,D,E,F** → A, G, H, I, L, M, N, O, P, Q **A,J** → K **A** ↠ L,M,N,O
Person	person id (A), name (B), city (C), phone number (D), email (E), gender (F), data of birth (G), age (H), height(I), weight(J), BMI(K), clinic code patient(L), clinic code physician(M), other health issues (N), title in clinic(O)	**A** → B, C, D, E, F, G, H **A,L** → I, J, K, N **A,M** → O
Clinic	clinic code (A), name (B), address (C), phone number (D), email (E)	**A** → B, C, D, E

Table 4.4: OSA entities with their attributes and FDs

Entity	Primary key / candidate key
Source	(source identifier, channel number) as AC
Recording	(recording id, sample timestamp) as AJ, and (source identifier, channel number, id person collects recording, id person own recording, recording timestamp) as BCDEFJ
Person	(person id, clinic code patient, clinic code physician) as ALM
Clinic	(clinic code) as A

TABLE 4.5: Classified entities with their primary/candidate keys

1. T := X
2. As long as T changed, if there is a FD A $\rightarrow$ B in FD set, where A is a subset of T, T:=T$\cup$B
3. X+ := T

Source

A and C are not presented in the right hand side of the two FDs. After computing AC+, all of attributes are retrieved. There is no need to expand AC when it is a candidate key. Therefore, AC is the only candidate key of this entity[1].

Recording

J are not presented in the right hand side of the FDs. After computing J+, none of attribute are retrieved. Expand J with A, after computing AJ+, none of attribute are retrieved. Therefore, AJ is a candidate key of this entity. Repeat the expanding process, an other candidate key can be found, which is BCDEFJ. Therefore, this table has two candidate keys which are AJ and BCDEFJ.

Person

A, L, and M are not presented in the right hand side of the FDs. After computing ALM+, all of attribute are retrieved. Therefore, ALM is the only candidate key of this entity.

Clinic

All of other attributes depend on A, therefore A is the only candidate key of this entity.

Table 4.5 presents the results after performed the key-finding algorithms.

4.3.2 Normalization

Normalization is essential for relational database tables in terms of integrity, maintainability and performance. After classifying, identifying attributes and relationships for entities, which is a tuple in a table in relational database, the table may produce redundant data when the entities are stored in a database system, if the table is not normalized. Moreover, it may take long time to search some particular rows due to the redundant data. Update and delete are extremely expensive when the redundant

[1]To be easy to follow, entity, relation, and table are used alternatively.

Normal form	Definition	Characteristic
First normal form (1NF)	- All columns contain only atomic values - Each column can have only one value (or nil) for each row	- Repeating groups in a table are eliminated - A primary key is used for identifying each set of related data
Second normal form (2NF)	is 1NF, with FD: X $\rightarrow$ A, where X is a set of attributes and A is an attribute. One of the following must be hold: - X is a super key - A is a key-attribute - X is not a subset of any candidate keys	the same as 1NF, plus: - All non-key attributes are fully FD on the primary key
Third normal form (3NF)	is 2NF, with FD: X $\rightarrow$ A, where X is a set of attributes and A is an attribute. One of the following must be hold: - X is a super key - A is a key-attribute	the same as 2NF, plus: - There is no transitive FDs
Boyce-Codd normal form (BCNF)	is 3NF, with FD: X $\rightarrow$ A, where X is a set of attributes and A is an attribute. One of the following must be hold: - X is a super key	the same as 3NF, plus: - All FDs are super keys
Fourth normal form (4NF)	is BCNF, with MVD: X $\twoheadrightarrow$ Y, where X is a set of attributes and Y is an other set of attributes. One of the following must be hold: - X is a super key	the same as BCNF, plus: Y is a subset of X, or X and Y together form the whole set of attributes of the relation

Table 4.6: An overview of normal forms

data are large. These costs are caused by the fact that the database management system must do an update or a delete operation for each of the redundant data. A method to break a large redundant table into many compact, non-redundant tables to avoid the high costs of redundant data, is called normalization. After normalization, the database would become much more reliable and efficient. Table 4.6 present a short summary of famous normal forms which are derived from the INF3100(Database System course)(Evgenij_Thorstensen, 2017) lecture. Although 3NF eliminates most of the anomalies in databases, there are still some anomalies when a table has multiple overlapping candidate keys. The BCNF can eliminate these anomalies, but the BCNF can not solve the multivalued dependence problem. Therefore, 4NF is chosen as the highest form for the design. For each table (entity) in Table 4.4, a procedure to check and normalize these table is presented as follow(Munthe-Kaas, 2017):

For each entity with its FDs and MVDs:

1. For each of MVDs X $\twoheadrightarrow$ Y, decompose R to R1(XY) and R2(XZ) where Z is all attributes in R and not in XY.

2. For each of FDs, BCNF checking and decomposing processes as follow(Munthe-Kaas, 2017): For each entity with its FDs:

2.1. All candidate keys are listed.

2.2. All the right hand side with multiple attributes must be split into atomic FDs (only one attribute on the right hand side).

2.3. For each atomic FD X→A, check the FD with the rules in Table 4.6 to find the normal form of this FD.

The normal form of the entity (relation) is the lowest normal form of the FDs. Once the normal form of the table is found, if it is not at the desirable normal form, decomposition can be taken place as follow(Munthe-Kaas, 2017):

Assume there is a relation R with FDs F that breaks BCNF:

1. If $X \rightarrow A$ breaks BCNF, compute X+, then decompose R into S and T, where S:=X+, T:=X∪(R-X+). 2. Repeat 1 with the new relations (in this case are S,T) until all relations are decomposed to BCNF.

Source

There is no MVD in this table. Candidate key of the table is AC.

This table has two FDs, which are **AC** → DEFGHIJK and **A** → BL.

After split, the table has $F = \{AC \rightarrow D, AC \rightarrow E, AC \rightarrow F, AC \rightarrow G, AC \rightarrow H, AC \rightarrow I, AC \rightarrow J, AC \rightarrow K, A \rightarrow B, A \rightarrow L\}$. - $A \rightarrow B$: The left hand side of this FD is not a super key, therefore, it breaks BCNF, This FD is neither 3NF, because the right hand side of FD is not an attribute in candidate key. The FD is not 2NF, since the right hand side is a subset of candidate key. Hence, the FD is 1NF.

Since 1NF is the lowest normal form, there is no need to scan the other FDs to find the normal form of the table. The normal form of a table is the lowest normal form of its FDs. Therefore, the normal form of table Source is 1NF.

Let a new relation R1 = A+ = (ABL); an other relation R2 = A∪(R-R1) = (ACDEFGHIJK). After the composition, A is a primary key of table R1(ABL) as SensorSource(source identifier, source name, source type), and AC is a primary key of table R2(ACDEFGHIJK) as Channel(source identifier, channel number, channel name, metric, transducer type, physical maximum, physical minimum, digital maximum, digital minimum, EDF reserved compatible). Therefore, this composition is in BCNF.

Recording

This table has one MVD which is A $\twoheadrightarrow$ LMNO. This MVD can be split into two tables R1 and R2. R1 contains all attributes from the MVD, that is ALMNO; R2 contains the attributes from the left hand side of the MVD and the attributes that do not exists on the right hand side of the MVD, they are ABCDEFGHIJKPQ. R2 is named Recording; R1 is named Annotation. To decompose the table to 4NF, the Recording need to be decomposed to BCNF with respect to the FDs.

Table Recording has two candidate keys which are AJ and BCDEFJ. The table has three FDs, which are **A** → BCDEFGHILMNOPQ, **BCDEF** → AGHILMNOPQ, and

AJ → K.
After split, the table has $F = \{A \rightarrow B, A \rightarrow C, A \rightarrow D, A \rightarrow E, A \rightarrow F, A \rightarrow G, A \rightarrow H, A \rightarrow I, A \rightarrow L, A \rightarrow M, A \rightarrow N, A \rightarrow O, A \rightarrow P, A \rightarrow Q, BCDEF \rightarrow A, BCDEF \rightarrow G, BCDEF \rightarrow H, BCDEF \rightarrow I, BCDEF \rightarrow L, BCDEF \rightarrow M, BCDEF \rightarrow N, BCDEF \rightarrow O, BCDEF \rightarrow P, BCDEF \rightarrow Q, AJ \rightarrow K\}$. - $A \rightarrow B$: The left hand side of this FD is not a super key, therefore, it breaks BCNF. This FD is neither 3NF, because the right hand side of FD is not an attribute in candidate key. The FD is not 2NF, since the right hand side is a subset of candidate key. Hence, the FD is 1NF.
Since 1NF is the lowest normal form, there is no need to scan the other FDs to find the normal form of the table. Therefore, the normal form of table Source is 1NF.
Let a new relation R1 = A+ = (ABCDEFGHIPQ); an other relation R2 = A∪(R-R1) = (AJK). After the composition, A and BCDEF are primary keys of table R1, AJ is primary key of R2, and A is primary key of Annotation(ALMNO). Since A depends on BCDEF, and LMNO depends on A, the composition is satisfy the 4NF.
The Recording table is split into R1(ALMNO) as Annotation(recording id, annotation onset, annotation duration, annotation timekeeping, annotation text), R2(ABCDEFGHIPQ) as Record(recording id, source identifier, channel number, id person collects recording, id person own recording, recording timestamp, recording description, frequency, pre-filtering, used equipment, EDF reserved), and R3(AJK) as Sample(recording id, sample timestamp, sample value).
However, in Annotation table, a many-to-many relationship between records and annotations need to be solved. The many-to-many relationship exists, because there is a transitive relationship between records for a source an its annotations. That is, one source has one set of annotations for all channels belong to the source at a specific timestamp. Furthermore, a record is defined by a source id, channel is, patient id, physician id, and timestamp. Therefore, one record has many annotations, and one annotations belong to many records. To solve the many-to-many relationship, Annotation table need to be split into RecordAnnotation(record id, annotation id) and Annotation(annotation id, onset, duration, timekeeping, annotation text).

Person

There is no MVD in this table. Candidate key of the table is ALM.
This table has three FDs, which are **A** → BCDEFGH, **AL** → IJKN, and **AM** → O. After split, the table has $F = \{A \rightarrow B, A \rightarrow C, A \rightarrow D, A \rightarrow E, A \rightarrow F, A \rightarrow G, A \rightarrow H, AL \rightarrow I, AL \rightarrow J, AL \rightarrow K, AL \rightarrow N, AM \rightarrow O\}$. - $A \rightarrow B$: The left hand side of this FD is not a super key, therefore, it breaks BCNF, This FD is neither 3NF, because the right hand side of FD is not an attribute in candidate key. The FD is not 2NF, since the right hand side is a subset of candidate key. Hence, the FD is 1NF.
Since 1NF is the lowest normal form, there is no need to scan the other FDs to find the normal form of the table. Therefore, the normal form of table Source is 1NF.
Let a new relation R1 = A+ = (ABCDEFGH); an other relation R2 = A∪(R-R1) = (AIJKNLMO). After the composition, A is a primary key of table R1, but AL and AM are not primary keys of table R2. Therefore, the composition need to be

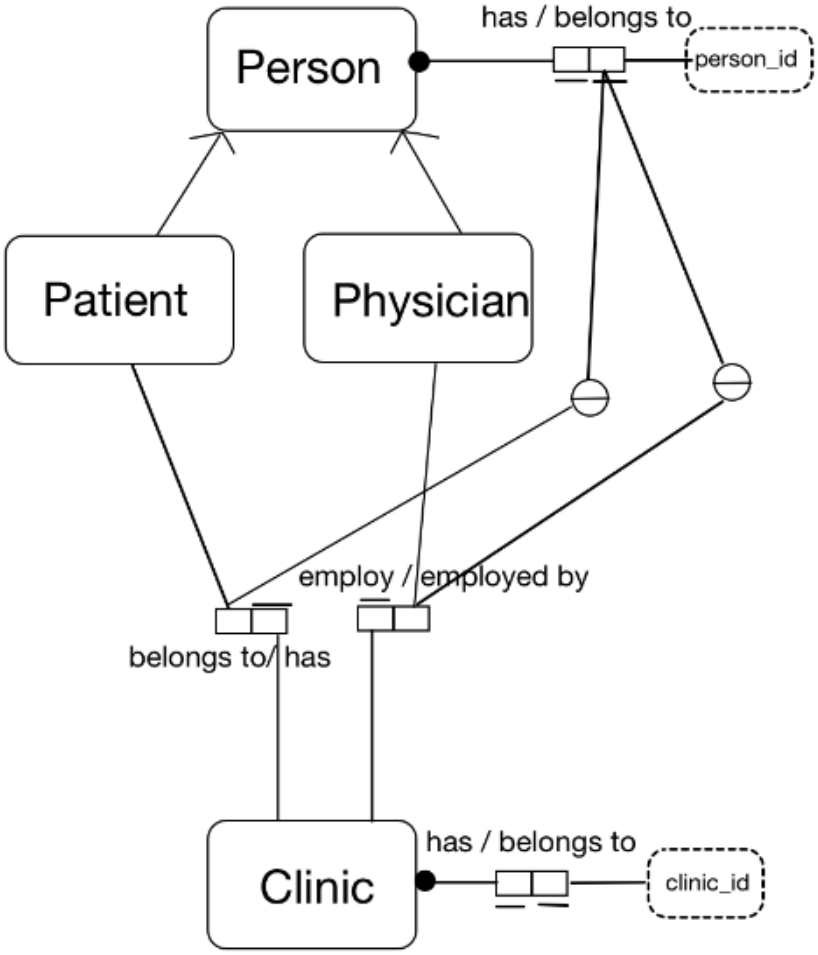


FIGURE 4.7: Logical model of the OSA database - Person and Clinic

repeated on R2. Let a new relation R3 = AL+ = (ALIJKN); an other relation R4 = AL∪(R2-R3) = (ALMO). After the composition, AL is a primary key of table R3, and AM is a primary key of table R4. However, R4 and R3 is sub-object of R1, hence, R4 does not need to take L. Therefore, the decomposition is in BCNF with relations R1(ABCDEFGH) as Person(person id, name, city, phone number, email, gender, data of birth, age), R3(ALIJKN) as Patient(person id, clinic code patient, height, weight, BMI, other health issues), and R4(AMO) as Physician(person id, clinic code physician, title in clinic).

Clinic

Clinic has only one FD which is also the primary key of the relation. Therefore, it is automatic in BCNF.

Figure 4.7 and 4.8 presents a summary of the logical model of the database design

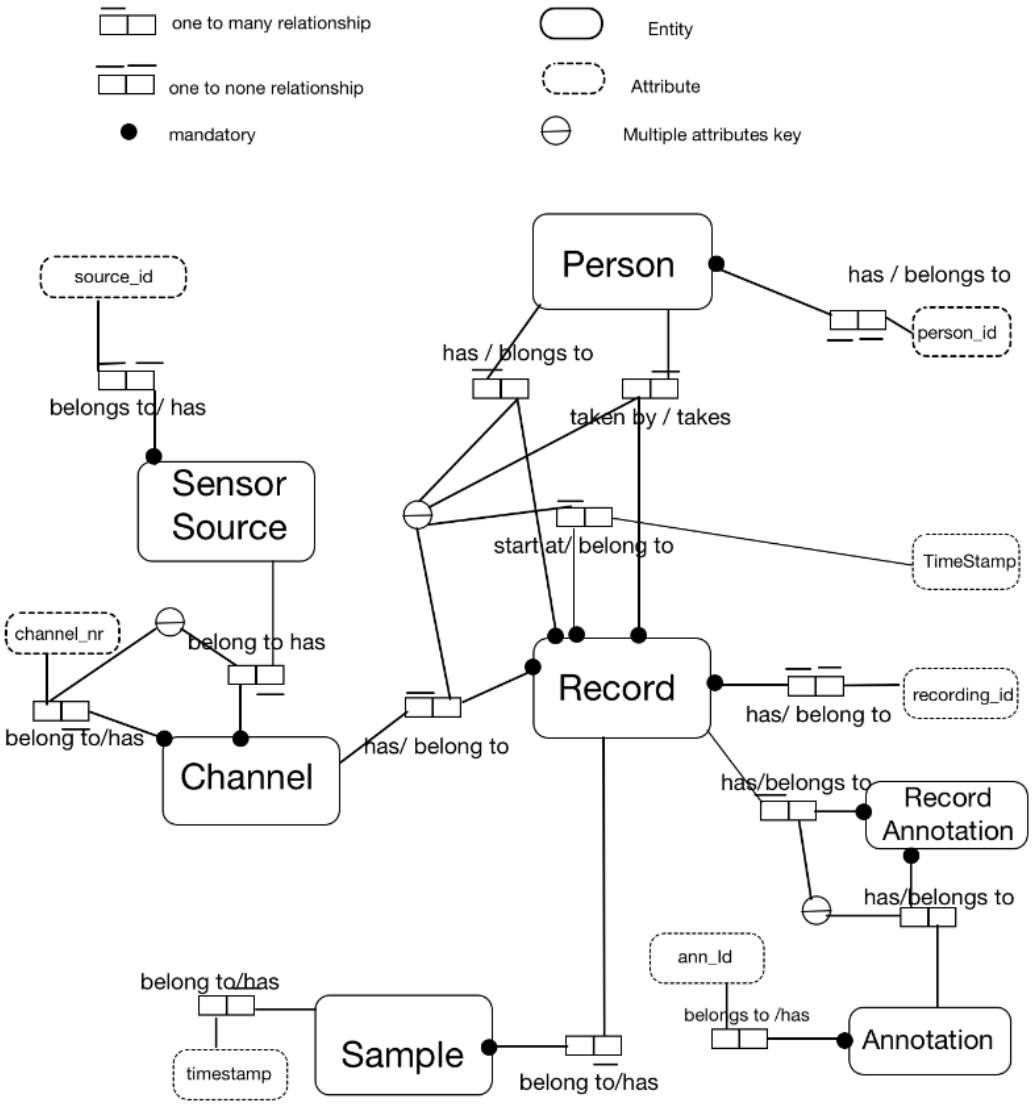


FIGURE 4.8: Logical model of the OSA database - Source and Recording

after decomposition. In this figure, non-key attributes are hidden to maximize the abstraction.

4.4 Physical data modeling

The logical data model, which is presented in Figure 4.7 and 4.8, is a platform independent model. This model can be implement on a workstation computer or even mobile platform as long as these platforms have a database management system. Since this thesis chooses the Android operation system as a platform to implement the design, the SQLite database manage system is chosen for modeling the physical data. An overview of SQLite database management system is discussed in Subsection 4.4.1. Subsection 4.4.2 presents the transformation of logical data model to physical data model.

4.4.1 SQLite database management system

As presented in SQLite site(SQLite, n.d.), SQLite is an embedded SQL database engine. It does not support client-server process as other database management systems do. However, other SQL database elements that are needed for this design are fully supported. They are multiple tables, indices, triggers, and views. In SQLite, all the mentioned elements are included in a single file on disk. In terms of opening file for doing analysis and modifying, using a database management is much better than using buffer and file functions offered by a programming language.
SQLite has five data storage classes that can be used for the physical data modeling.

- NULL: The value is a NULL value.

- INTEGER: The value of this type is a signed integer. Depend on the magnitude of the stored value, it could be one, two, three, four, six, or eight bytes in size.

- REAL: The value of this type is a floating point value following the IEEE floating point number. This type is eight bytes in size.

- TEXT: Unlike the other database management systems, SQLite has only one type for storing a text string. The text could be encoded as UTF-8, UTF-16 big-endian, or UTF-16 little endian.

- BLOB: The value of the type is BLOB(Binary Large OBject) stored in exactly the same format as it is provided to the database.

In SQLite, the term of “datatype” is not used, but “storage class”, since data are dynamically stored in the system. Therefore, the data need to be converted before

Person		
Columns	Data type	Explanation
person_id	TEXT	person id can contains alphabetic character, not null
name	TEXT	name is a string, can be null for anonymous
city	TEXT	city is a string, can be null for anonymous
phone_nr	TEXT	phone number can contains +, can be null if not have
email	TEXT	email is a string, can be null if not have
gender	TEXT	gender can be a character or sometime a string, must be not null for later analysis
date_of_birth	TEXT	date of birth is a string, must be not null for later analysis
age	INT	age is an integer, can be null

Table 4.7: Transforming Person into SQLite table

storing, or using. However, at an abstract level, the two terms present data type. Hence, these terms can be used interchangeably.

4.4.2 Transforming the logical data model to SQL

As presented in Figure 4.7 and 4.8, there are nine entities that must be transformed into tables. The names of these entities are also used for the corresponding tables in SQLite. Attributes of the entities become columns in the tables, and the relationships now become the foreign keys. Tables 4.7 to 4.16 present the attributes for each table with the explanation for each chosen type.

Patient		
Columns	Data type	Explanation
person_id	TEXT	foreign key to Person, not null
clinic_code	TEXT	foreign key to Clinic, not null
patient_code_p	TEXT	Patient code in clinic, not null
height	REAL	it must be floating number, can be null
weight	REAL	it must be floating number, can be null
BMI	REAL	it must be floating number, can be null
other_health_issues	TEXT	it is a string, can be null

Table 4.8: Transforming Patient into SQLite table

Physician		
Columns	Data type	Explanation
person_id	TEXT	foreign key to Person, not null
clinic_code_f	TEXT	foreign key to Clinic, not null
employee_id	TEXT	Physician code in Clinic, not null
title_in_clinic	TEXT	it is a string, can be null

Table 4.9: Transforming Physician into SQLite table

Clinic		
Columns	Data type	Explanation
clinic_code	TEXT	it can contains text, not null
name	TEXT	it is a string, can be null
address	TEXT	it is a string, can be null
phone_nr	TEXT	phone number can contains +, can be null if not have
email	TEXT	it is a string, can be null

Table 4.10: Transforming Clinic into SQLite table

SensorSource		
Columns	Data type	Explanation
source_id	TEXT	it can contains text, not null
source_name	TEXT	it is a string, can be null
source_type	TEXT	it is a string, can be null

Table 4.11: Transforming SensorSource into SQLite table

Record		
Columns	Data type	Explanation
recording_id	INT	unique long int from the system when created
source_id	TEXT	foreign key to SensorSource, not null
channel_nr	TEXT	together with SensorSource, it is foreign key to Channel, not null
person_collect	TEXT	foreign key to Person, not null
person_owner	TEXT	foreign key to Person, not null
timestamp	INT	Unix time when the recording is started, not null
description	TEXT	can be the applied position on the body, can be null
frequency	REAL	collected at frequency, can be null
used_equipment	TEXT	other used equipment name, can be null
EDF_reverved	BLOB	reserved for EDF, byte array which may contains EDF+C, EDF+D, etc., can be null

Table 4.12: Transforming Record into SQLite table

RecordAnnotation		
Columns	Data type	Explanation
recording_id	INT	foreign key to Record, not null
annotation_id	INT	foreign key to Annotation, not null

Table 4.13: Transforming RecordAnnotation into SQLite table

Annotation		
Columns	Data type	Explanation
annotation_id	INT	unique long int from the system when created, not null
annotation_onset	INT	when the annotation started, not null
annotation_duration	INT	how long an annotation last
annotation_timeKeeping	INT	if it is the first annotation in annotation lists
annotation_text	TEXT	text of the annotation

Table 4.14: Transforming Annotation into SQLite table

Sample		
Columns	Data type	Explanation
recording_id	INT	foreign key to Record, not null
sample_timestamp	INT	Unix time, when it is created, not null
sample_value	REAL	it is a float number, not null

Table 4.15: Transforming Sample into SQLite table

Channel		
Columns	Data type	Explanation
source_id	TEXT	foreign key to SensorSource, not null
channel_nr	INT	numerical order, not null
channel_name	TEXT	name of channel, not null
metric	TEXT	metric, not null
transducer_type	TEXT	it is string, can be null
physical_maximum	REAL	physical maximum, can be null
physical_minimum	REAL	physical minimum, can be null
digital_maximum	INT	digital max, can be null
digital_minimum	INT	digital min, can be null
pre_filtering	TEXT	applied filtering on this channel, can be null
EDF_channel_reserved	BLOB	EDF reserved

Table 4.16: Transforming Channel into SQLite table

SQLite code for creating the Channel table is presented in Listing 4.1, creating code for the other tables can be found in Appendix D.

Listing 4.1: SQLite code for creating table Channel

```
1     CREATE TABLE CHANNEL(
2             SOURCE_ID                TEXT NOT NULL,
3             CHANNEL_NR               INT NOT NULL,
4             CHANNEL_NAME             TEXT NOT NULL,
5             TRANSDUCER_TYPE          TEXT,
6             METRIC                   TEXT,
7             PHYSICAL_MAX             REAL,
8             PHYSICAL_MIN             REAL,
9             DIGITAL_MAX              INT,
10            DIGITAL_MIN              INT,
11            PREFILTERING             TEXT,
12            EDF_CHANNEL_RESERVED     BLOB,
13            PRIMARY KEY (CHANNEL_NR ,SOURCE_ID),
14          FOREIGN KEY(SOURCEID) REFERENCES
               TABLE_SENSOR_SOURCE(SOURCE_ID)
15           );
```

Figure 4.9 presents the final physical database model of the database system.

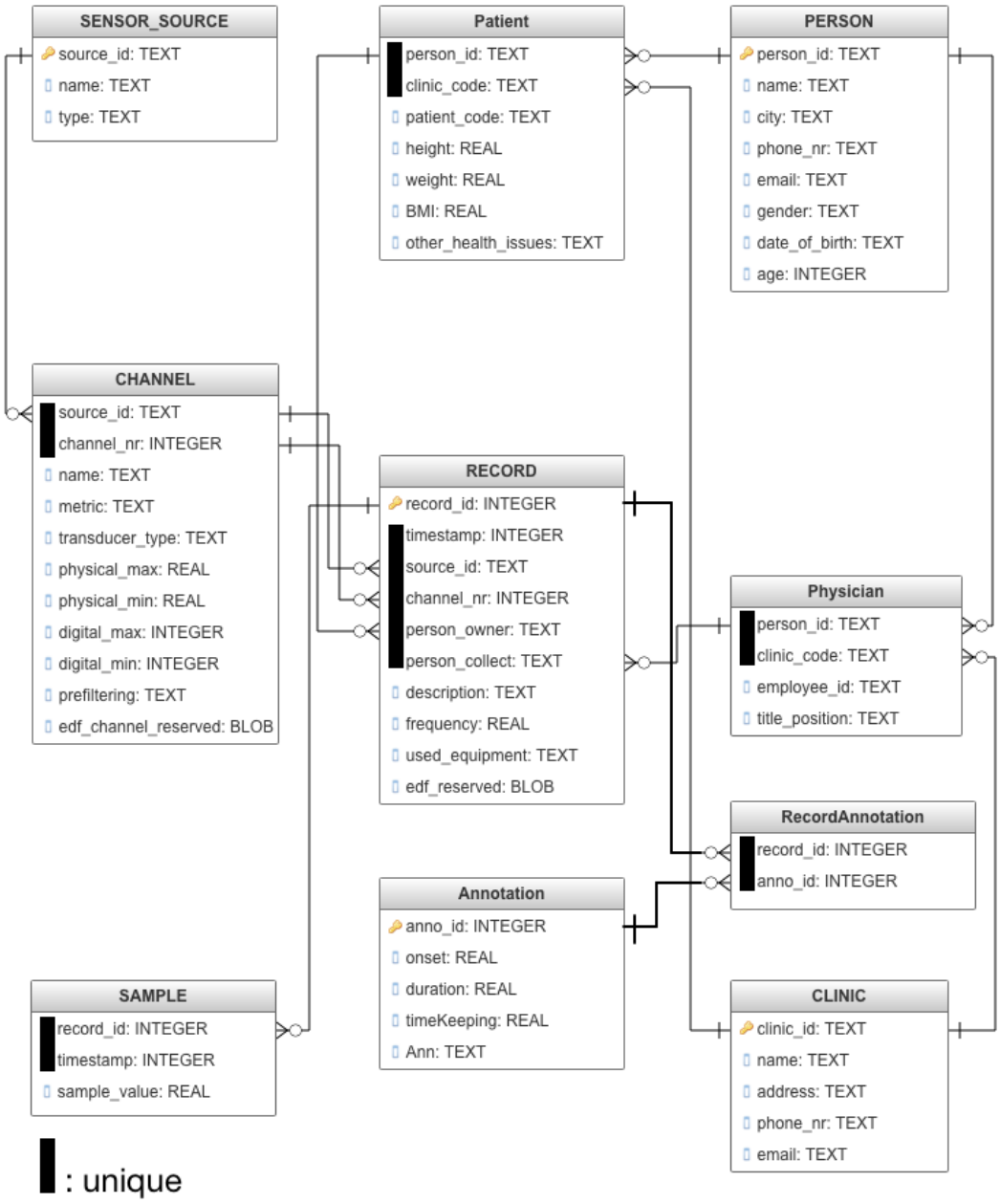


FIGURE 4.9: Physical database model

Chapter 5

Implementation

Patients, physicians, and researchers do not only want to store OSA signal into a relational database system, but also want to do data analysis based on the collected data. Computing ability and storage capability of mobile devices have substantially increased, and the users would like to store and perform data analysis on these devices. Therefore, a database system application on a mobile platform for the designed database model in Chapter 4 needs to be taken into considering. The implementation mainly focuses on developing sensor wrappers on Android operating system for the designed database, and writing SQLite codes to do data analysis. Hence, other functions such as GUI interactions between users and the application, visualizing data, etc., are of minor importance in this thesis. It is to say, not all possible features are implemented, therefore, the implementation is a proof-of-concept. This chapter presents a traditional software engineering approach, in which, the waterfall process model approach is mainly used for developing the application. It is because the process activities can be separately presented (requirements specification, software design, implementation, etc.). An incremental development approach is useful when the requirements change frequently, and new functions need to be added to the previous version. Hence, it is not suitable for the OSA database system, since the data requirements are stable. At the time of the writing, the OSA data are not stored in any relational databases. Instead, each clinic has their own file format for storing the data. The users can either use the provided tools from these clinic, or ask for a general format file, like EDF format, in order to analyze the data. Therefore, reuse-oriented software engineering approach cannot be used, since the database application must be developed from scratch rather than integrated into an exist system.
In this chapter, functional and non-functional requirements of the users are carefully analyzed. These requirements are presented in Section 5.1; they are supplementary to the discussed requirements in Chapter 4. Section 5.2 presents an abstract model of the database system application and architectural models with respect to real-time wrapper and EDF wrapper. Section 5.3 presents an Android specific implementation of the discussed models.

User requirements definition	System functional requirements	System non-functional requirements	Structured specifications
The application must reuse CESAR acquisition tool to collect data from BITalino.	1. System must open a port for that CESAR acquisition tool can connect and send data. 2. System must follow CESAR package formats for that the data can be correctly collected. 3. System must let the users fill out the requirement fields for patient and clinic before storing a record into database system. 4. System must support multiple connections.	- Product: usability, performance, space, reliability - Organization: Android operative system (6.0) - External: must follow the protection of personal data of patient	- Input: metadata and data packages from CESAR - Source: BITalino - Output: store metadata and data to database system, and may plot them to graph - Place: fragment server application and fragment real-time visualization
The application must support importing and exporting EDF files.	1. The user can freely choose a EDF file to import, and import progress must be showed. 2. The user can partially import an EDF file by click stop button, in case they do not want to wait. 3. System must support fully or partially export. That is, the user can choose from time – to time when exporting. 4. The user can choose which channels they want to export.	- Product: usability, performance, space, reliability - Organization: Android operative system (6.0) - External: must follow the protection of personal data of patient	- Input: EDF header and EDF data record - Source: EDF file - Output: store/export EDF header and data record to database system/EDF file - Place: fragment EDF reader and fragment EDF exporter
Data analysis can be perform by using the application.	1. The system must support raw query, in which the users can write SQL queries to retrieve whatever they want. 2. The system must provide some mining functions to detect OSA signal.	- Product: usability, performance, space, reliability - Organization: Android operative system (6.0), SQL query language	- Input: data in database system - Source: database system - Output: result from SQL, or OSA detection - Place: mining fragment
Collected data could be visualized in real-time data, or replay from the stored data.	1. The user can visualize data on a graph view. 2. Channels can be freely choose to visualize to do comparison.	- Product: usability, performance, space, reliability - Organization: Android operative system (6.0)	- Input: BITalino or database - Source: BITalino or database - Output: graphic view - Place: fragment real-time/replay visualization
Annotations could be added manually to a certain source.	Annotations could be manually added and stored while visualizing sources.	- Product: usability, performance, space, reliability - Organization: Android operative system (6.0)	- Input: data in database - Source: database - Output: annotations from users are stored in database system - Place: fragment replay visualization

Table 5.1: A summary of the database system application requirements

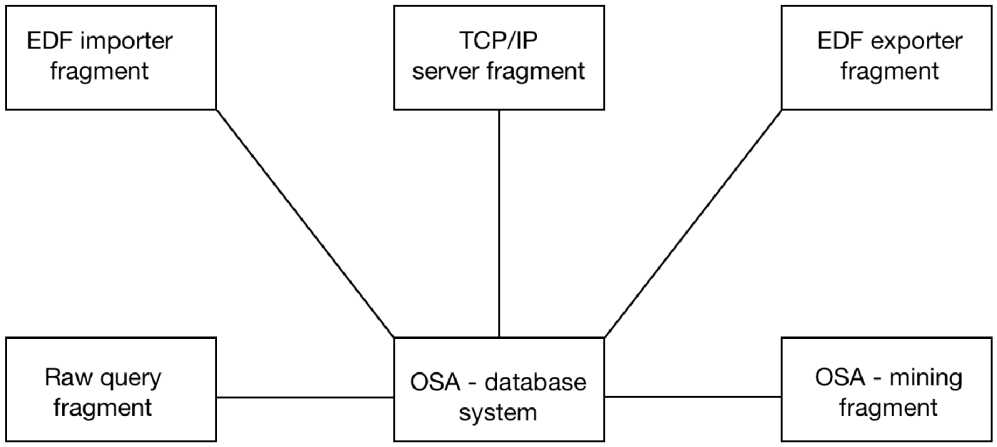


FIGURE 5.1: The context of the OSA database system application

5.1 Requirements for database application

This section presents and analyzes the specific requirements of the users as well as the database application system. The analysis results provide the foundation for designing the data model and the implementation of the application.
User requirements are usually presented as statements. These statements are in natural language, and are about services that the system is expected to provide to the users, and constrains for the services. In addition, system requirements present a list of system requirement specifications for each user requirement statement. In short, the users define the requirements, while the system specifies in detailed the services it provides, inputs and outputs, functional and non-functional requirements, etc., for each of defined requirement.
Functional requirements are statements of services the system should provide, how the system should react to particular inputs, and how the system should behave in a particular situation (Ian, 2017). By contrast, non-functional requirement are constraints on the services or functions offered by the system. They include timing constrains, constrains on the development process, and constraints imposed by standards (Ian, 2017). Table 5.1 presents the requirements of the database system application. In which each user requirement defines the services the system must provide. Correspondingly, the system requirements clearly explain how the system must behave to satisfy the user requirement. These system requirements are presented in forms of functional requirements and non-functional requirements, and structured specifications of the requirements. The “Place” in the structure specifications column presents which function groups the requirements belong to. By grouping requirements in a group of functions, it is easier to design and implement. The function groups are illustrated in Figure 5.1. The first two and the last three requirements in Table 5.1 are respectively corresponded to “what it must contain” and “what the database is to be used for” presented in the database modeling chapter. In other words, the database application needs to be implemented two wrappers that collect data from

CESAR acquisition tool and EDF/EDF+ file formats, then the collected data can be used for analyzing, visualizing, or modifying. Since the main focus of the thesis is collecting data and data analysis, visualizing and modifying are added as helper functions.

The first two user requirements illustrate that the system must support collecting samples from BITalino by using the CESAR acquisition tool, and importing EDF/EDF+ data. As presented in Chapter 3, for collecting samples from the CESAR acquisition tool, the system first establishes a TCP/IP connection to the tool, then the metadata and data packages are sent to the system. The structure of these packages are well documented, and have been discussed in Chapter 3. The user can also use multiple acquisition tools to collect samples, hence the application must support multiple connections as well. Since CESAR acquisition tool does not send the information of the clinic and patient, the information must be manually filled by the user before recording samples. The CESAR acquisition tool provides data in real-time, therefore the system must operate with respect to the given non-functional requirements presented in Table 5.1. Likewise, for importing and exporting EDF/EDF+ files, the system must follow the structure of the EDF/EDF files which are discussed in Chapter 3. Since the files can be very large, the system must satisfy the non-functional requirements when importing and exporting.

As mentioned earlier, visualizing and modifying are not the main focus of the thesis. They are added to the database system application as helper functions to help the user to have a better view on the collected data. Therefore, they are implemented as proof-of-concept and "enough for using". For the analysis requirements, a raw query function is useful when analyzing the data. However, in term of security, this is quiet dangerous action. In the top 10 vulnerabilities, SQL injection stands on the top of the list(veracode.com, 2017). In this case, the risk does not come from stealing of sensitive information, or compromising the database. It is dangerous if the researchers accidentally perform queries that can result the database system corrupted, such as deleting a column in a data table, drop a table, etc. Filter out vulnerable queries is a topic for researchers who are interested in database security. Hence, to filter out vulnerable queries is not in scope of the thesis. An assumption is made that the users have the knowledge on database system, and they do not perform any vulnerable queries. The system must also provide some of the possible mining algorithms that are used for detecting OSA.

5.2 Database application modeling

As presented in Figure 5.1, the context of the OSA database system consists of importing, exporting and analyzing data. Data sources, in which the system collects data from, can be divided into two groups. Sources that connect to the database system via TCP/IP protocol are real-time sources, while, non-real-time sources are from EDF/EDF+ files. Data analysis is only performed on the stored data. Currently,

the system does not support real-time analysis, since the goals of the system is to collect raw data for future analysis. That is, the collected data are used as the inputs for many different analysis algorithms like the possible mining methods presented in this thesis. However, real-time OSA data analysis is good to be considered, and an exciting topic for researchers who are interested in online analytical processing.
Subsection 5.2.1 presents real-time wrapper modeling, in which, some real-time attributes are taken into considered, and how the TCP/IP server fragment is modeled. Subsection 5.2.2 presents the modeling for non-real-time wrapper, in which the EDF importer fragment and EDF exporter fragment are carefully modeled.

5.2.1 Real-time wrapper

When choosing the appropriate real time sensor sources, several factors should be considered, such as the quality of signal, mobility, how many channels can it observe, which protocol it uses to send data sample, etc. The BITalino platform is chosen as sensor source after carefully considering the pros and cons of it in Chapter 3.
In terms of real-time data stream source, the system has to deal with data stream management problems. The thesis targets at a solution to store the OSA bio-signals, and it does not address data stream management system issues, where the queries need to be apply on the data stream. Instead, some general real time factors need to be seriously considered when designing the database system. These factors are arrival rate, timestamp, physical resource, one-time read data, data stale or imprecise, and unpredictable data arrival. When the incoming rate is high, it might be a problem to store all the data, because of the limitation of the mobile platform. Even on a stationary computer, storing data streams could be a problem and needs to be considered.
To choose a suitable solution to manage the arrival rate of the data stream, it is good to review and discuss how the data stream management system (DSMS) deal with real-time data stream problems. Due to the unboundedness of the data stream, it is essential to capture the stream into small slices which is called windows. To manage the data stream, a DSMS uses window models, in which the models are based on the direction of movement of the endpoints, that are fixed window, sliding window, and landmark window. As the name of the window models, the fixed window has a fixed amount of samples or time interval. The sliding window contains the data from now up to a certain range in the past. The landmark window, on the other hand, contains the data from the beginning until now. The window size can be either physical/time-based or logical/count based. Figure 5.2 presents an overview of the fixed window and sliding window. When a sample arrives, it is updated either immediately (eager processing) or when the window is full (lazy/batch processing).
Tasks of the system are to store the raw data samples as well as to display these samples to a graph view with respect to the users' requirements. To ensure that the tasks are well performed, the system uses two buffers which functioned as windows in DSMS. It is to say, the first buffer is a fixed window, count based, and batch

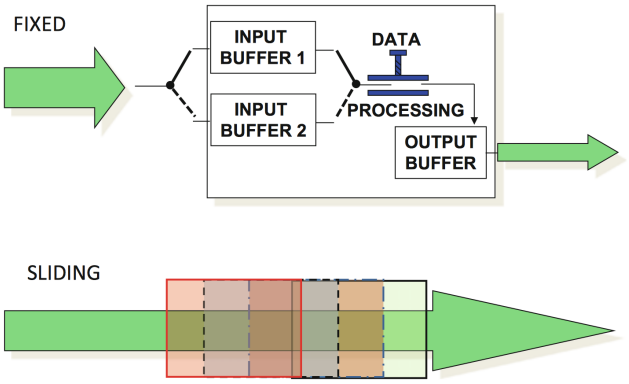


FIGURE 5.2: Example of fixed and sliding windows (Emanuele_Panigati, 2017)

processing, that is used for collecting samples for the database. The second buffer is a sliding window, count (or time) based, eager processing, that is used for collecting samples for graphic view. By using these two buffer, the system ensures that the overhead of writing to database is minimized (by using batching processing), while the performance increases (by using eager processing).
If the order of the samples plays an important role for later analysis, the timestamp must be explicit (timestamp from data source). Otherwise, the implicit timestamp (system time) can be used. The CESAR acquisition tool provides the timestamp in each sample object. However, the timestamp is pre-converted and needs to be handled before using for a plot view. It is an overhead to convert the timestamp for each sample. However, it is quite easy to change some code in the acquisition tool such that it sends a raw timestamp (Unix timestamp). Therefore, the explicit timestamp is considered a better solution than the implicit timestamp with respect to the accuracy of the arrival samples.

Server thread

The collector of the CESAR acquisition tool offers two methods for the database application to collect data from it. The collector can either save data to a text file or send them to a given server IP and port address by using the TCP/IP protocol. Since the database system application wants to have real-time data from BITalino, it must open a port to collect data. As presented in the requirement section, the system must support multiple connections, because the user may use multiple sensor sources to monitor the body. Therefore, a multiple threads system must be implemented. Each thread manages one sensor source. The main thread therefore just waits for connections, creates and hands in necessary information to the new created thread, then starts the new thread. The main thread must have a way to manage the created threads for that the users can choose a source they want to interact with from the connected list. A Unified Modeling Language (UML) activity diagram is

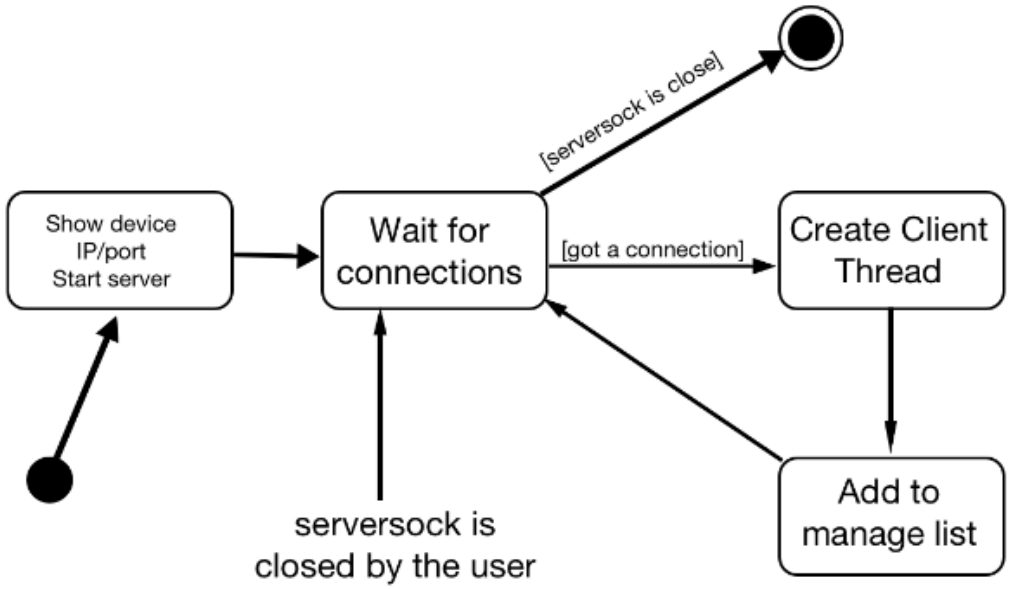


FIGURE 5.3: Process model of server thread

used for illustrating how the main thread works as presented in Figure 5.3. UML is a famous and widely used modeling language, however, a short explanation on the used annotations is needed to make the figures easier to understand. In an UML activity diagram, a filled circles indicates the start of a process. Activities are presented by rectangles with round corners. Arrows present the flow of work between activities. Annotations on the arrow indicate the condition when the work flow is taken. A filled circle inside another circle indicates the end of the process.

Client thread

Most of the jobs of the wrapper are handled in the client thread. Once a client thread is created, it waits for data to arrive, then pushes the data to the database or adds it to a graphical view if these actions are flagged. Data packets from CESAR acquisition tool are well discussed in Chapter 3, in which a metadata packet is sent first to identify the sensor source, then the source keeps sending its data via the connection between the database system application and the acquisition tool. As explained in the real-time characteristics, the client thread must share two buffers with the other threads. The first buffer is used for storing samples to the database, and the second buffer is used for showing samples on a graphic view. However, these buffers are initialized only if the corresponded flags are flagged. That is, arrival samples are dropped if the users do not want to store or visualize them. Figure 5.4 presents a possible implementation of the client thread by using a UML activity diagram.

As illustrated in the requirements, the system must be reliable and have a good performance. That is, no samples data are lost under recording, and the visualization must not freeze. To satisfy these requirements, two different buffer management methods are used. A buffer, which is used for storing samples, maintains a list of fixed number of samples (it is to say, a record fragment), and a thread. The thread

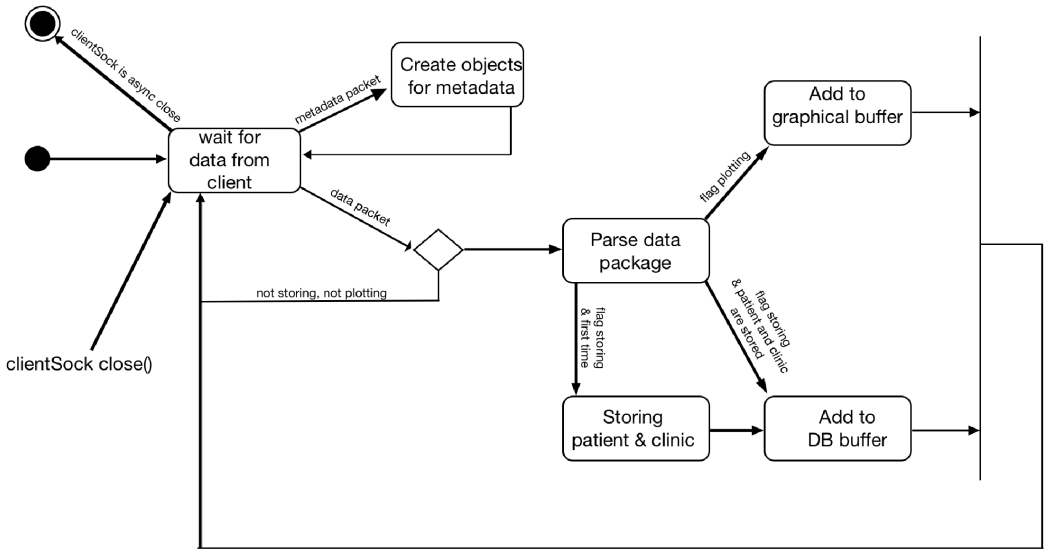


FIGURE 5.4: Process model of client thread

takes a full record fragment to store into the database, or waits for available record fragments if the list is empty. Since SQLite can perform about 50,000(SQLite, 2017b) insert statements per second, while the maximum number of samples BITalino can delivery is 1000 samples per second (1000Hz), the algorithm for this buffer should therefore satisfy the non-functional requirement (reliable). The second buffer can be implemented by using the algorithm from the Producer and Consumer problem, in which the client thread is the producer, and a thread that updates the graphic view is the consumer. However, this buffer is used as a sliding window, therefore a simpler solution can be used. That is, each time the client thread adds a sample to a buffer list, it removes the oldest sample if the buffer is full. After that, a graphic view thread is notified to refresh the plot view.
Figure 5.5 presents how the sever thread, client thread, database, and graphic view connect to each other's.

5.2.2 Non real-time wrapper

Diverse data sources are essential in research. Using multiple sources of data in research often produce more accurate results and more objective values than using a single data source. There are many trustworthy sources that can be used for OSA data analysis. One of the sources that is chosen in this thesis are the Physionet sensor databases, which have been described in Section 3.1.2. The other non-real time source that the system collects data from, is the NOX-T3 sensor source system. Theoretically, this source can be used as a real time sensor source. Although Nox-medical has an Android application to collect samples from their devices, which is only supported by NOX-A1-PSG at the time of this writing, the NOX-T3 device neither provides any API for a mobile platform, nor any documents that describe how to collect samples from the device as BITalino does. As mentioned earlier, it is very difficult to manage

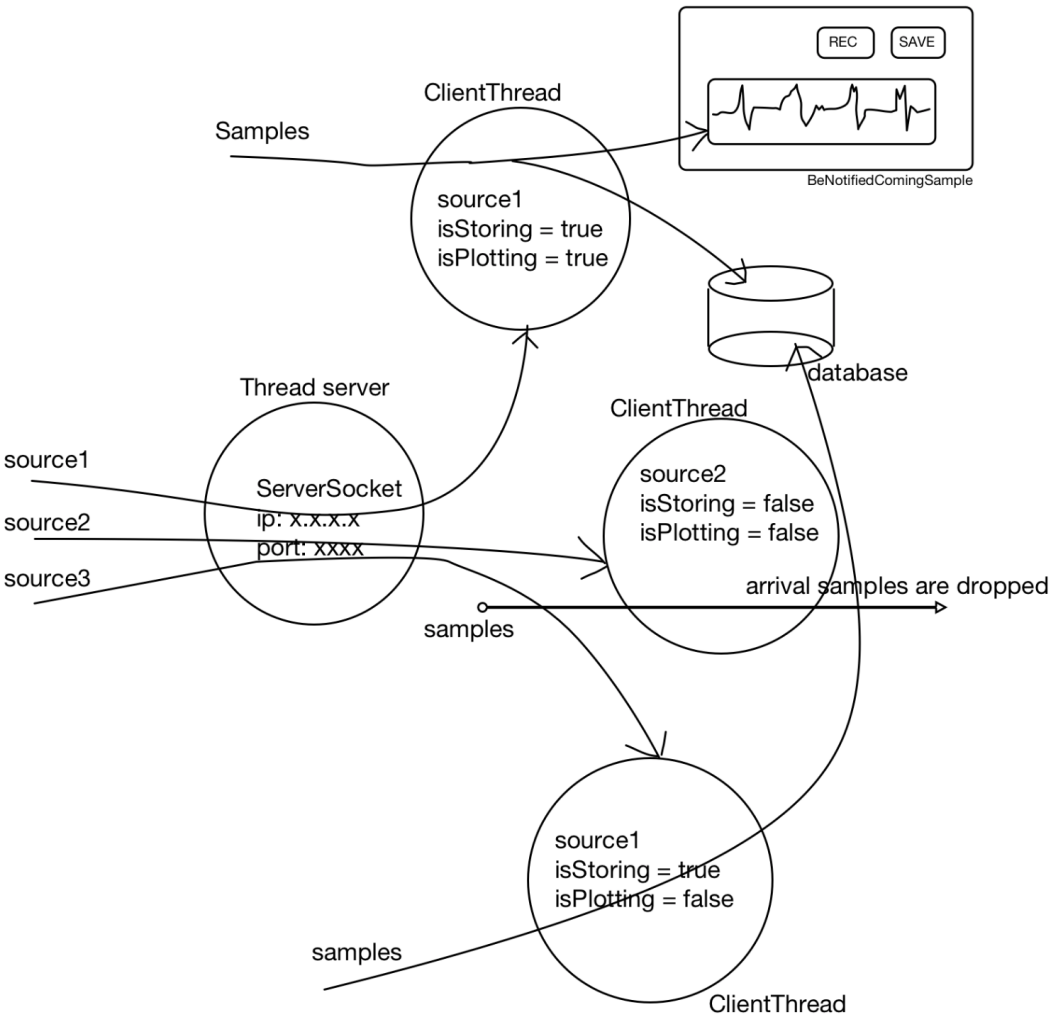


FIGURE 5.5: High level design of real-time wrapper

diverse sources when they have their own format. Fortunately, the problem is solved by using an EDF or an EDF+ file format to share data between source owners. The formats are fully described in Section 3.2.2 with respect to why they are introduced, what information these file contain, and how to use them. Therefore, the system is designed in the way that is open for all of the sensor sources, as long as these sources can export their data to an EDF or EDF+ file format. In other words, the system only accepts source files in EDF form.
An EDF file can be very large, and can excess the main memory size. Hence, to satisfy the performance and robustness, the system should neither keep all the data in memory, nor call the database insert function for each sample. The problem could be solved by using the idea from real time sensor source. In other words, the system uses the concept of a window model with lazy update (batch processing) to solve the memory problem with the non-real time source.
Physionet databases and the NOX-T3 sensor source can be used as non-real-time sources, because both of them provide a function to export their bio-signal data to EDF. "mit2edf" is a function from physiotools provided by Physionet. This function is used for converting between EDF and WFDB-compatible formats. NOX-T3 provides a graphic, step by step, and user friendly way to export their data to EDF.

EDF importer

As introduced in Chapter 3, EDF is one of the standardized data formats for bio-signals that is used for storing and exchanging multichannel biological and physical signals. There are many tools and open source codes which can be used for reading a EDF file. Different tools have different goals when reading the EDF file. Most of them parse the samples to a graphical view and an annotations list, the others try to convert samples into text files that contain information for each channel and the record. EDF browser and EDF library(teuniz.net, 2017) are one of the most famous used tools to view and parse EDF files. The performance of the tools is quite good since they are written in C code. Another open source tool that can be used for parsing EDF files to text files is Java-parser for EDF format(MIOB, 2017). As the tool named, it is written in Java code, hence, the performance when parsing EDF file is poorer compared with EDF library. Since EDF/EDF+ file formats are well documented, it is easy to write a parsing tool. As discussed, different tools have different purposes when parsing EDF files. Since the one of the main goals of EDF file format is used for exchanging biological data, the EDF files need to be parsed into the received system data structure. Many parsers try to load the whole EDF file into main memory before converting. As discussed, the EDF file can be extremely large, the parsers therefore crash; Java-parser for EDF format is one them.
There is no need to reinventing the wheel rather using them in a smart way. In the designed database system each bio-object is stored in a separate table, and the EDF importer can use the functions in an EDF library to read the EDF files. The information, which are read from EDF, are stored into the corresponding tables. In

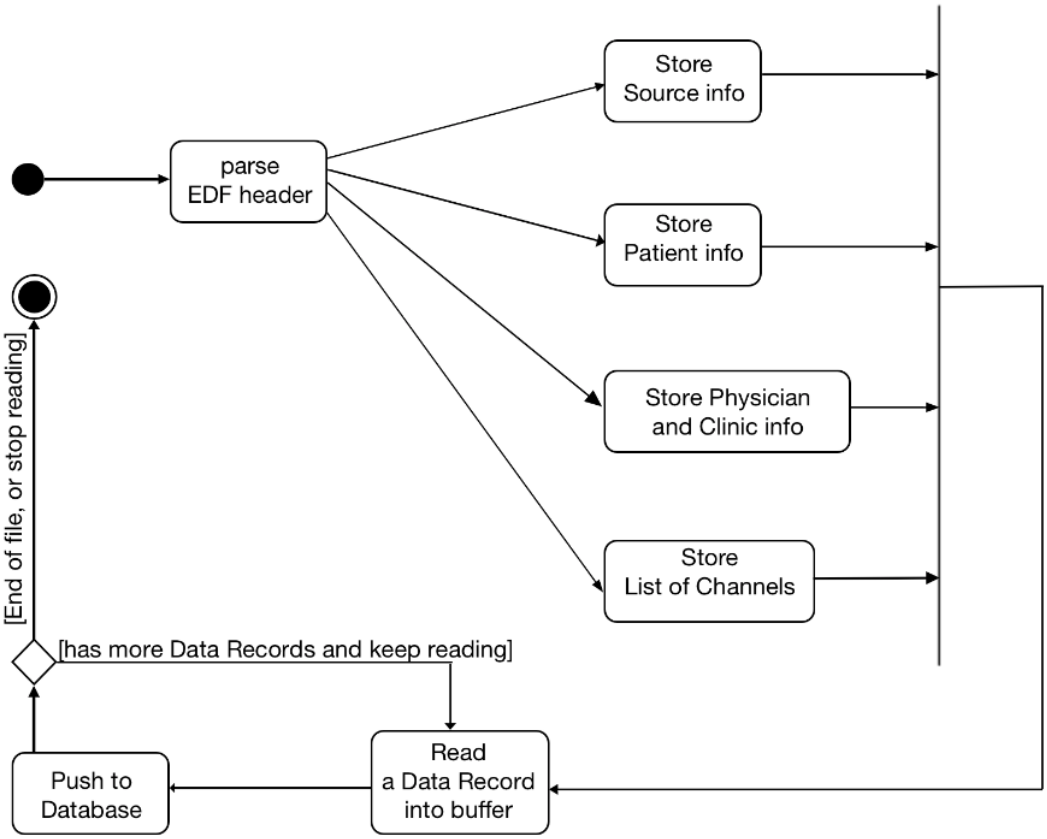


FIGURE 5.6: Process model of EDF importer

case the used library tries to read the whole EDF file into memory, an optimization, which is multiphase read, need to be used. Figure 5.6 presents a UML activity diagram which explains how an EDF file can be read into the database without memory problem.

EDF exporter

Sharing data is essential in research. Therefore, the system must export its data into a standardized data format for bio-signals, e.g in an EDF file. However, the system targets to be implemented on a mobile platform, where resources are limited. Exporting data and saving it in external storage places is vital to satisfy non-functional requirements, where the collected data must not be lost when the storage capability of the mobile devices is exceeded. There is no need to export the whole source of data to a EDF file, some of channels or samples are exported for special needs. For example, if there is a project, in which researchers or physicians need only samples from ECG channel, it does not make sense if the EDF file contains samples for the other non-relevant channels. Furthermore, if a project needs to analyze all samples which are collected during nighttime, the added daytime samples waste not only the storage, but also the time to parse the EDF file when using. Therefore, the system must let the users choose which channels and periods they want to export. As discussed in the non-functional requirements for EDF importer and exporter, the information of the

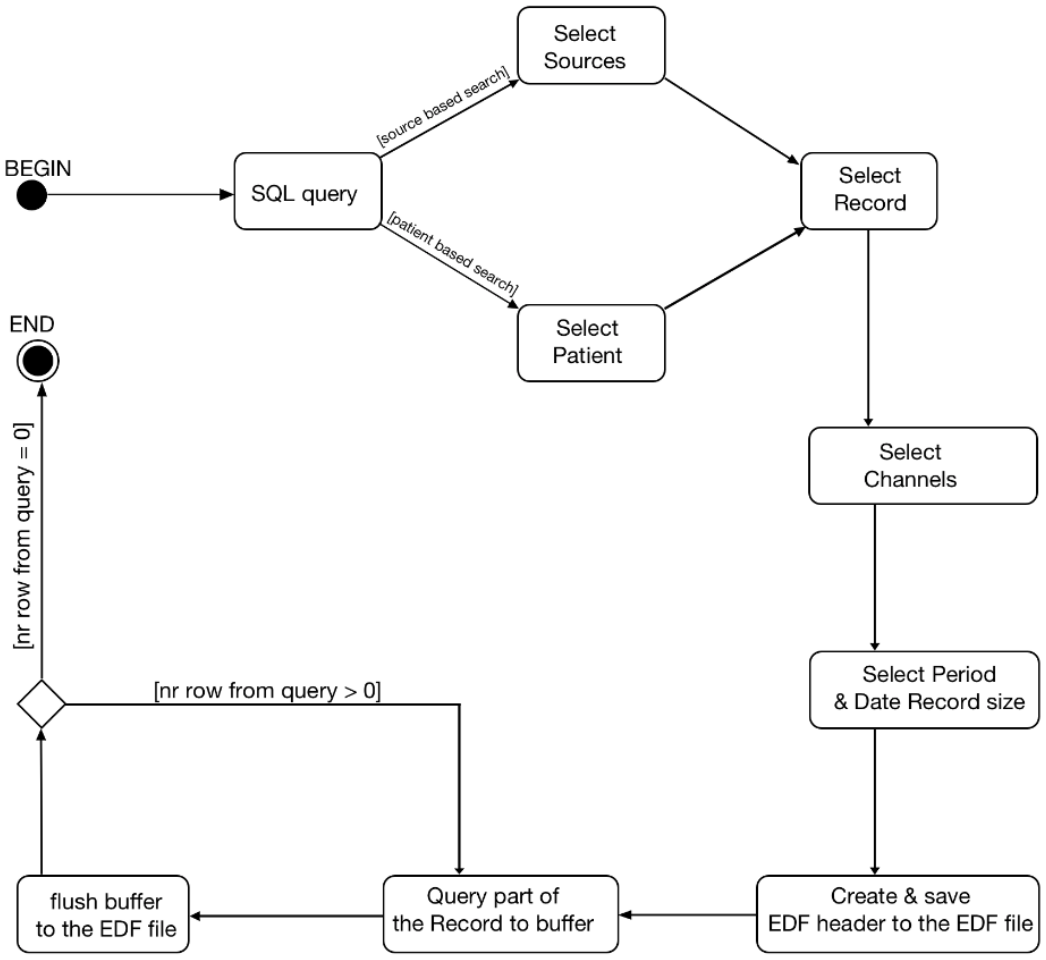


FIGURE 5.7: Process model of EDF exporter

patient must follow the protection of personal data law. That is, the patient must be exported as anonymous, otherwise there must be an agreement of the patient. Figure 5.7 is a UML activity diagram that illustrate how data are exported into a EDF file.

5.3 Database application implementation

Based on the database and the database application modeling, a chosen software platform must be operated on mobile devices, and must support features that the designs require. Both Android and iOS operative system are the good candidates for the position. They all support threads, SQLite, TCP/IP connection, Bluetooth, etc. However, it is difficult to say which one is better, it depends on the user favorites. Since the implementation is a proof-of-concept, it is no need to deeply argue why choosing iOS or Android as long as they meet the requirements. Android platform is chosen for the designs in the thesis for some reasons. First of all, according to netmarketshare.com(netmarketshare.com, 2017), there is 66.71% devices installed Android operative system compared to 29.55% devices installed iOS. Moreover, applications for Android are written by using Java programming language, which is very easy to be

read and understood by many developers. Nonetheless, it is easy to synchronize and test with the CESAR acquisition tool, because the tool is written for Android. Beside satisfying the functional and non-functional requirements, the implementation also focuses on the graphic user interface (GUI). It is natural that people have a better feeling when they interact with icons and symbols compared to text. By using a GUI, the system can avoid asking input from users by letting the users choose via provided input or default values, because typing text on a mobile platform is a cumbersome task.

Subsection 5.3.1 presents a short discussion on how to manage multi-threaded database accessing in SQLite database system on Android platform. Subsection 5.3.2 presents how to implement the real-time wrapper model on the chosen platform for collecting data from CESAR acquisition tool. Subsection 5.3.3 presents the implementation of non-real-time wrapper, in which the EDF import and export functions are implemented accordingly to the abstract designs in Subsection 5.2.2. Subsection 5.3.4 presents a simple implementation, in which the collected data can be visualized on a graphical graph view.

5.3.1 SQLite and Multi-threaded database accessing

In SQLite, one database can be shared by multiple processes at the same time. By using read/write locks, SQLite can control the ways processes access the database. The processes can perform read operations at the same time, but only one process performs write operations at any moment in time. From Version 3.5.0, SQLite manages locks internally to avoid data corruption. Hence, several threads can use a single SQLite connection simultaneously. That is, the developer does not need to manage the concurrence between threads when they access the shared database. However, balancing database workloads between threads need to be considered, because when a thread wants to write to the database, it locks the entire database file for the time it uses for writing. A statistic on the relative number of devices running a given version of the Android platform from Google presents that more than 99,9% devices running an Android version with API 10 or better (Android, 2017b). According to the dependence between Android API and SQLite version(Android, 2017a), SQLite 3.6 comes with Android API 8; the higher the Android API is, the better SQLite version it probably has. Therefore, most of current mobile phones have the SQLite version better than 3.6 which supports the internal database-level locks to avoid database corruption. However, database accessing can be failed if each process has its own connection to the database file. It is because the SQLite does not support synchronization between multiple connections. When one connection is in use for writing, the database rejects the other modifying activities from other connections. As a result, the database management does not update changes of the other connections. To use multiple threads for maximizing database performance is not benefited, since there is only one modifying connection at a specific time.

Threads can be used to maintain a shared connection for different data sources, or

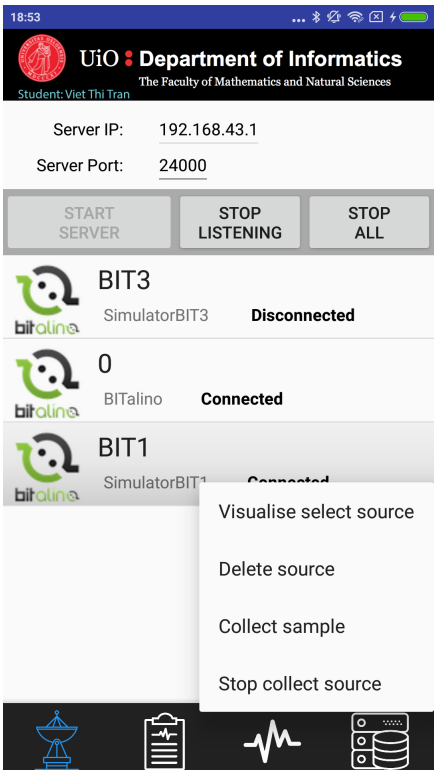


FIGURE 5.8: Graphic user interface of the wrapper for CESAR acquisition tool

different data using purposes. These threads take turn using the database connection. By sharing the connection, the application makes sure that all threads can correctly update their data into the database. In this thesis, a helper object OSADataBaseManager is implemented in such a way that it is transparent for threads who using it. That is, a thread can initialize an instance of the object, then it can get the SQLiteDatabase via the openDatabase() method of the instance. After performing a database operations, the thread can ask the instance for closing the database connection. From the view of the thread, it is logic when a thread that opened a database connection for some tasks closes the connection. However, the OSADataBaseManager instance creates and maintains only one SQLiteDatabase connection for all threads. The connection is created whenever there is at least one thread that wants to connect to the database, and closed when there are no database requests. Listing 5.1 presents how to manage the shared database connection when working with multi-threaded database access.

```
1     private int mOpenCounter;
2         private static OSADataBaseManager instance;
3         private static OSADBHelper mOSADBHelper;
4         private SQLiteDatabase mDatabase;
5
```

```
6           public static synchronized void initializeInstance
               (OSADBHelper helper) {
7               if (instance == null) {
8                   instance = new OSADataBaseManager();
9                   mOSADBHelper = helper;
10              }
11          }
12
13          public static synchronized OSADataBaseManager
               getInstance() throws Exception{
14              if (instance == null) {
15                  throw new Exception(OSADataBaseManager.
                       class.getSimpleName()
16                           + "␣call␣initializeInstance(..)␣to
                               ␣initialize␣instance.");
17              }
18              return instance;
19          }
20
21          public synchronized SQLiteDatabase openDatabase()
               {
22              mOpenCounter++;
23              //If it is the first time
24              if(mOpenCounter == 1) {
25                  mDatabase = mOSADBHelper.
                       getWritableDatabase();
26              }
27              //else just return the opened instance
28              return mDatabase;
29          }
30
31          public synchronized void closeDatabase() {
32              //We do not want to close the DB while the
                   other use it
33              mOpenCounter--;
34              if(mOpenCounter == 0) {
35                  //REAL CLOSE
36                  mDatabase.close();
37              }
38          }
```

Listing 5.1: SQLite connection management

5.3.2 CESAR wrapper

As presented in the high level design of real-time wrapper, the wrapper must implement two kinds of threads. The first one is used for maintaining a list of connected sources that not only from the CESAR acquisition tool, but also from other acquisition tools, as long as the sources follow the package interfaces that are discussed in Chapter 3. The second one is used for maintaining the connection between the application and a certain sensor source. Figure 5.8 presents the GUI of the wrapper. In the figure, the application provides the IP number and the port it listens to. There are two active sources (with status "connected") and one inactive source (with status "disconnected") in the figure. By long clicking on a source, a list of functions, which the user can interact with, is shown. Each component that is presented in the figure is equivalent to a function the wrapper needs to provide. The functions are divided into two groups; the first group consists of starting server, stopping listening, stopping all client threads and disconnect the server, managing the current connected list, and providing functions for that users can interact with the current connected list. These functions belong to server thread which is implemented in **ServerFragment.java** in the implementation code. The second group consists of collecting data from a specific source, forwarding data to graphical view (if requested), managing batch processing for storing data into the database. These functions belong to client management thread which is implemented in **ClientThread.java**.

ServerFragment.java

As Figure 5.3 presented, the application must get the IP address of the device and show it in the GUI. It can be done by checking all the network interface devices, and finding the IPv4 from the interfaces. Users are freely to choose a port number which must be bigger than 1024, because port smaller than 1024 are system ports(IANA, 2017). To make it easy for the user, a default port is provided before the application is started. Once everything is initialized, the user can click on the "START SERVER" button to begin the listening process, which waits for sensor sources at the presented address and port number. When the user clicks on the "START SERVER" button, the button is disabled, and a server thread is created. The procedure when the button is clicked is illustrated as following:

```
1          server = new ServerSocket(portNr);
2                  loop:
3          1.  wait for connections from clients; if get
              connection, go to Step 2
4          2.  create a ClientTread object with necessary
              parameters
5          3.  add the ClientThread into the managing list
6          4.  start the thread and go to Step 1
```

```
7                   if server socket is closed (by clicking
                       either "STOP "LISTENING or "STOP "ALL
                       buttons), the server is shutdown, and
                       the "START "SERVER button is enabled.
```

LISTING 5.2: Pseudo code to manage connections

Listing 5.3 presents how the abstract design and the procedure are translated into coding in Android. With respect to the GUI, "Toast" is used for posting a notification such that the user knows some events have happened. Other logics and codes for managing GUI are not the main focuses of the thesis, and are therefore not discussed in depth. These codes can be found in the included project folder.

```
1  serverPort = Integer.parseInt(txtServerPort.getText().
       toString());
2  final Context CONTEXT = getContext();
3  Thread server = new Thread(new Runnable() {
4          @Override
5          public void run() {
6              try {
7                  //Create a server socket object, bind it
                       to server_port
8                  sockServer = new ServerSocket(serverPort);
9                  //Multi clients management
10                 while (true) {
11                 //Accept the client connection, then give
                       it to ServerThread with client socket
12                     Socket socClient = sockServer.accept()
                           ;
13                     final String clientIP = socClient.
                           getRemoteSocketAddress().toString()
                           ;
14                     serverUpdateUI.post(new Runnable() {
15                         @Override
16                         public void run() {
17                             Toast.makeText(CONTEXT,"Got␣
                                   connect␣from:␣"
18                             +clientIP,Toast.LENGTH_SHORT).
                                   show();
19                         }
20                     });
21                     ClientThread clientConnected = new
```

```
22                                      ClientThread(socClient,
                                           CONTEXT,serverUpdateUI
                                           , selv);
23                      addNewSource(clientConnected);
24                      clientConnected.start();
25                  }
26              } catch (IOException e) {
27                  serverUpdateUI.post(new Runnable() {
28                      @Override
29                      public void run() {
30                          Toast.makeText(getContext(),
31                          "SERVER␣IS␣SHUT␣DOWN",Toast.
                               LENGTH_SHORT).show();
32                      }
33                  });
34                  //WHEN sockServer close, it will be here
35                  sockServer = null;
36              }
37          }
38          });
39  startServer.setEnabled(false);
40  stopListening.setEnabled(true);
41  stopAll.setEnabled(true);
42  server.start();
```

Listing 5.3: Server management

For each of the sensor source in the connected list, the user can interact with the connected source via four provided functions, that are collecting sample, stopping collecting sample, visualizing, and delete the source from the list. With visualizing, if the source is not fully initialized, or disconnected, the user is not allowed to view the source. Otherwise, the application disconnects all client threads from the graphical graph, then connects the selected source to the graphical graph. It is because the graphical graph allows only one source to be visualized at a specific time. Any optimizations for the graphical graph are considered future works, since the thesis mainly focus on database and wrapper implementations. On the other hand, if the user clicks on “Delete source”, the application double checks if the user really wants to delete the source from the connected list. If this is a case, the selected source is forced to be disconnected, and is removed from the managing list. If it is collecting data for the database, all the data in the buffer must be flushed into the database to ensure that no data losing under collecting process.

The user can perform data collection on the sources with “connected” status. By clicking on one of them, the user is asked for information about patient, physician and

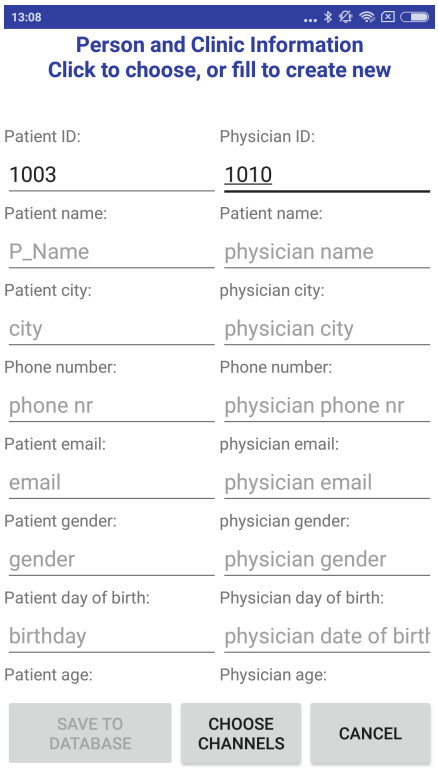


FIGURE 5.9: Form for getting user input for Patient, Physician, and Clinic

clinic information as presented in Figure 5.9. Since the patients are freely to choose the information they want to save, the fields in the form can be empty. However, the database need some information to manage the patient, physician and clinic, the user must at least provide information for the ID-fields. After that, the user must choose which channels to be stored, then by clicking on "SAVE TO DATABASE" button, the connected source is flagged as storing, and storing process is started. Stopping collecting is quite easy to implement, it can be done by flagging the selected thread as not storing, and ask the thread to flush the buffer into the database.

ClientThread.java

After a client thread is created, it immediately listens for the incoming data at the input stream of its socket. As presented in Chapter 3, the CESAR acquisition tool separates data packets that are sent to a client by using a newline character. The thread can get the arrival packages by calling readLine() on the input stream. For each line from the stream, the thread can parse the line into a JSON object, and the collecting process is initialized if the "type" of the JSON object is "metadata". Otherwise, the JSON object is parsed, then visualized and stored if "isPlotting" and "isStoring" are flagged. The UML activity diagram from Figure 5.4 in the high level design can be translated into specific implementation as following:

```
1          BufferedReader bf = get input stream from client
               socket;
```

```
2          Loop: as long as the client thread is not
               disconnected
3                    1.  read a line from bf
4                    2.  parse the line to JSON object
5                    3.  if object type is ""metadata, register
                         new sensor source, go to Loop
6                    4.  if object type is ""data, update
                        sample to graphical view and database
7                         then go to Loop
8          If the user clicks on "STOP "ALL button, or delete
            a source, the thread is flagged as
               disconnected, and the collecting process is
               ended.
```

Listing 5.4: Pseudo code to parse packets

The metadata and data package of the CESAR acquisition tool are modified in this implementation. The tool has a collecting frequency for each channel, but it does not include the frequencies in the metadata package. A modification is made by including these frequencies into the metadata package. In the data package, the timestamp in each package is converted from Unix timestamp into a string before sending. It is obviously not a good solution. There is overhead to convert timestamp for each sample, especially when the sending rate is high. Moreover, a string text (”HH:mm:ss.SSS” is 12 bytes) is more expensive to send compared to a Unix timestamp (8 bytes). If the collection is performed at midnight, the text timestamp is confusing the receiver, i.e. from 23:59:59.000 to 00:00:01.000, because the timestamp does not include the date. By sending a Unix timestamp, the problems are solved.
To register a new sensor source is necessary to parse the metadata package into SensorSource and Channel objects. These objects are not pushed into the database unless the user performs the collecting process. Likewise, update a sample is performed if at least one of the “isPlotting” and “isStoring” flags is flagged. Listing 5.5 presents how a sample is processed and updated in the system.

Listing 5.5: Update real-time samples

```
1  void updateSample(JSONObject jsonObj) throws JSONException
       {
2       if(!isPlotting && !isStoring) return;
3       long timeStamp = jsonObj.getLong("time");
4       //  CHANNELS DATA  Getting JSON Array node
5       JSONArray channelsData = jsonObj.getJSONArray("data");
6       BitalinoDataSample[] samples = new BitalinoDataSample[
           channelsData.length()];
7       for(int i = 0; i < channelsData.length(); i++){
8           JSONObject channelData = channelsData.
               getJSONObject(i);
9           String channel_nr = channelData.getString("id");
10          float channel_data = Float.parseFloat(channelData.
               getString("value"));
11          samples[i] = new BitalinoDataSample(timeStamp,
               channel_nr,channel_data);
12      }
13      //SEND TO DATABASE BUFFER OR PLOTTING
14      if(isStoring) manageIsStoring(samples);
15      if(isPlotting) manageIsPlotting(samples);
16 }
```

As presented is Listing 5.5, if neither plotting nor storing flags are flagged, the sample is dropped. Otherwise, samples are forwarded to graphical view and storing process if they are flagged. The graphical view maintains a sliding buffer to hold samples, and implements the interface ”BeNotifiedComingSample”. The interface has a function ”addNewSample(BitalinoDataSample[] samples)”. By calling this function, the graphical view is notified such that it can slide the buffer (if the maximum thread hold is reached), and refresh the GUI. In contrast to plotting, that needs to update samples immediately, storing process add new samples into a fixed buffer. Collected samples are flushed into the database by submitting the samples to a database update thread when the buffer is full. The client thread and the database update thread are synchronized by using the producer-consumer algorithm(Wikipedia, 2017b). However, a modification is made on the shared buffer for that the application can meet the real-time requirements. That is, the shared buffer is implemented as a linked list, and therefore is unlimited in size. The client thread does not need to wait for an empty slot in the buffer, such that it can submit the samples. It just adds the samples into the buffer, notify the database update thread, then continue to get new arrival samples. The database update thread waits for samples if the shared buffer is empty, otherwise it gets samples and initial a SQLite transaction to insert the samples into the database. By using transaction, INSERT statements that are surround with

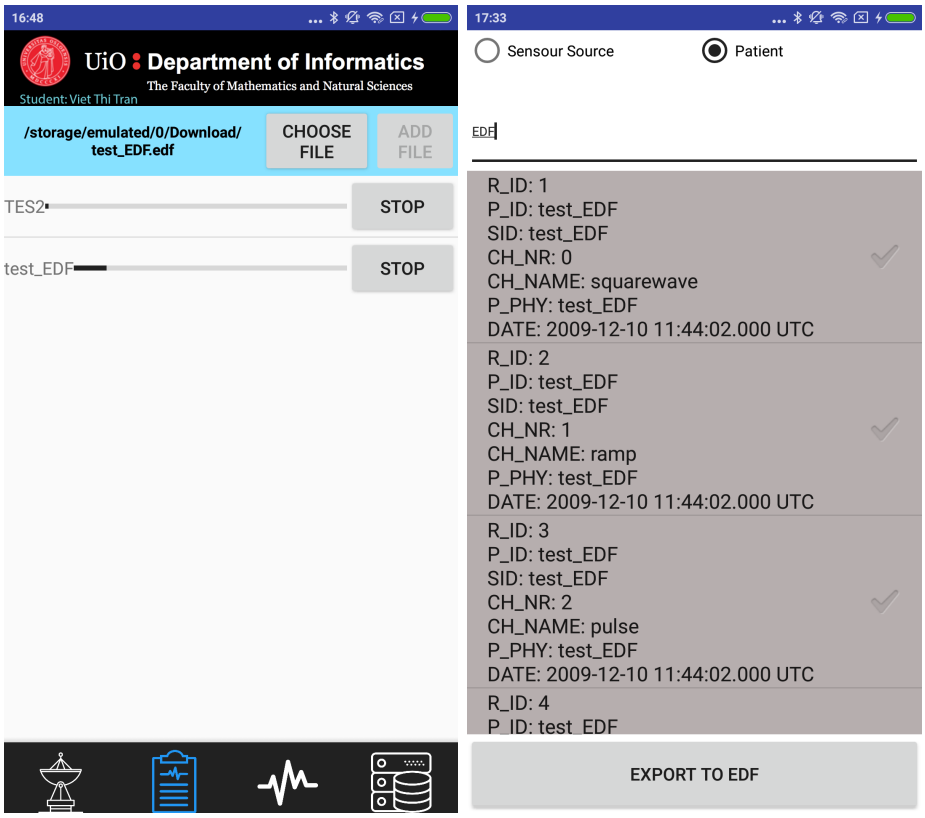
BEGIN and COMMIT are grouped into a single transaction instead of one transaction per INSERT statement. As a result, the performance of the system is increased. Listing 5.6 illustrates how to use transaction to store samples.

```
1  mDatabase.beginTransaction();
2  try{
3      for(Sample s : listSample){
4          ContentValues values = new ContentValues();
5          values.put(OSADBHelper.SAMPLE_RECORD_ID,s.getR_id
               ());
6          values.put(OSADBHelper.SAMPLE_TIMESTAMP,s.
               getTimestamp());
7          values.put(OSADBHelper.SAMPLE_VALUE,s.
               getSample_data());
8          mDatabase.insert(OSADBHelper.TABLE_SAMPLE, null,
               values);
9      }
10     mDatabase.setTransactionSuccessful();
11 }catch(Exception e){
12     e.printStackTrace();
13 }finally{
14     mDatabase.endTransaction();
15 }
```

Listing 5.6: SQLite insert samples transaction

5.3.3 EDF wrapper

The wrapper allows to import bio-signal data from Physionet databases into the database system. However, the wrapper accepts only EDF format, therefore the data from Physionet databases need to be exported to EDF format by using **"mit2edf"** function before it can be used by the system. Users can load multiple EDF files simultaneously. Figure 5.10(a) presents the GUI in which two EDF files are in parallel loaded with their status bar which shows how far the files have loaded. Users can partially load a EDF file, and can stop the loading process at any moment in time they want. Once a file is chosen, a thread is created to manage the file. At first, the header of the EDF file is parsed to a EDFHeader object. If the EDF file has the wrong format, the EDFHeader object is set to null, and therefore the programming stops reading the EDF file. That means, nothing is stored to the database if the file does not have the correct format. After parsing the EDF header, objects for Source, Patient, Physician, Clinic, Channel, and Record are created and pushed into the database. To avoid memory overflow, and not to hold the shared SQLite connect for long time, a fixed buffer is used for holding samples. That is, the samples are partially load



(a) EDF import GUI *(b)* EDF export GUI

FIGURE 5.10: EDF wrapper

into memory (buffer). When the buffer is full, the thread starts a SQLite transaction for the collected samples in the buffer. When the SQLite transaction is finished, the thread repeats the reading procedure until the EDF is totally read.

List 5.7 presents an overview of the EDF file read procedure. The wrappers also allow users to export information in the database to EDF or EDF+ file formats. As Figure 5.10(b) presents, the users can search all records based on either their ids, or the source ids they used for collecting data. After that, the users can choose which records they want to export to the EDF file. Annotations are depended on records, therefore, all annotations are queried into a buffer before the storing procedure for data records begins. As Figure 5.7 in the high level design presents, samples are partially queried into a buffer before flushing to EDF file. While writing the data records, the number of sample in the record must be set to minus one, because if something wrong happens while writing, the EDF file is registered as invalid file. When all data records are written to the file, the header of the file needs to be updated to valid the file. Listing 5.8 presents an overview of the EDF exporting procedure.

```
1  public void run(){
2      final String filePath = file_source.getFilePath();
3      try {
4          EDFHeader header = null;
5          InputStream is = new BufferedInputStream(
6                          new FileInputStream(new File(
                               filePath)));
7          //Parse Header to create new sensor source
8          header = EDFHeaderParser.parseHeader(is);
9          is.close();
10         if (header == null) {
11             sendMessageToHandler(FILE_IS_LOADED,
                   file_source.getIndex());
12             return;
13         }
14         createAndStoreSensorSource(header);
15         createAndSavePatientPhysicianClinic(header);
16         createAndSaveRecord(header);
17         createAndStoreChannels(header);
18
19         saveRecordFragmentAndSample(header);
20     }catch (Exception e){
21         e.printStackTrace();
22     }
23     sendMessageToHandler(FILE_IS_LOADED, file_source.
          getIndex());
24 }
```

Listing 5.7: EDF file reader

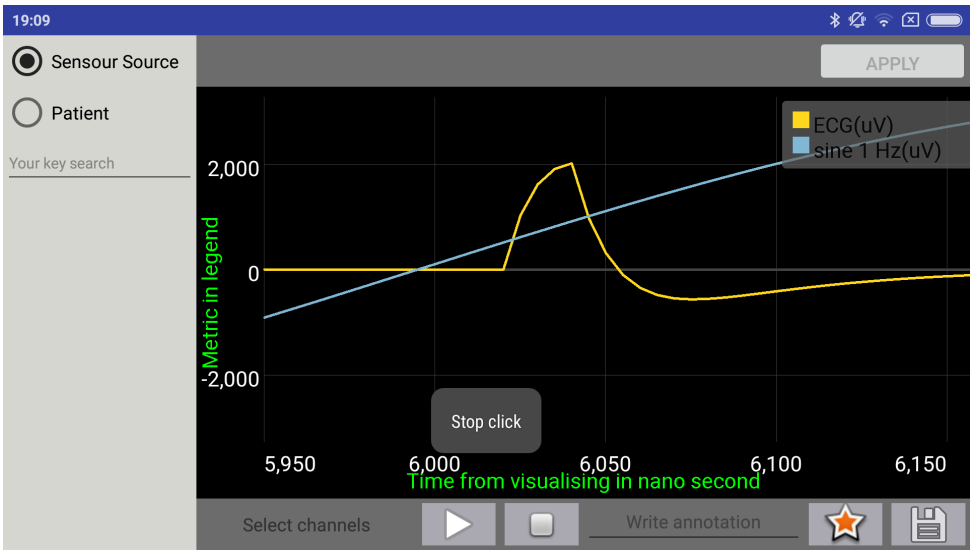


FIGURE 5.11: Replay samples from the database GUI

```
1  public void run(){
2      try{
3          raf = new RandomAccessFile(this.fileName, "rw");
4          buildEDFheader();
5          storeDataRecord();
6          //UPDATE TOTAL RECORD
7          raf.seek(0);
8          EDFWriter.writeEDFHeaderToFile(raf,edfHeader);
9          raf.close();
10     }catch (Exception e){
11         e.printStackTrace();
12     }
13 }
```

LISTING 5.8: EDF file writer

5.3.4 Real-time and non-real-time visualization

To represent samples, a graphical technique is used in the implementation, that is a line-based plot. Each sample is presented in the graph with respect to time and its values. Arriving samples update the graph dynamically. To avoid a memory problem, the implementation does not keep all samples in the buffer for plotting. A sliding window buffer is used to keep the presented samples. That is, when the buffer is full, the oldest sample is replaced by the new arrival sample. The implementation uses an open source module, which is Android Graph View(Grapview, 2017) for presenting the samples. To solving dynamic plotting, the source also uses a sliding window to avoid memory leaks. That is, before adding a data point to the graph, the buffer is checked if it is full, and the oldest data point is removed in case the max count is

reached. Listing 5.9(Grapview_code, 2017) presents the interface for appending new data points, and listing 5.10 presents how it is used in the implementation.

```
1 /**
2  * @param dataPoint values the values must be in the correct
      order!
3  *  x-value has to be ASC. First the lowest x value and at least
      the highest x value.
4  * @param scrollToEnd true => graphview will scroll to the end (
      maxX)
5  * @param maxDataPoints if max data count is reached, the oldest
      data
6  *  value will be lost to avoid memory leaks
7  * @param silent set true to avoid rerender the graph
8  */
9 public void appendData(E dataPoint, boolean scrollToEnd, int
      maxDataPoints, boolean silent);
```

Listing 5.9: appendData interface(Grapview_code, 2017)

In real-time visualization, the graph is passively waiting for other threads to update its buffer, and notify it when the buffer is updated such that the graph can refresh the GUI. Non-real-time visualization, in contrast, has to query data from the database, and presents the queried data on the graph. A user, therefore, can pause and play queried samples at any moment in time. However, the user cannot do it in real-time visualization. It is because if the feature is supported, some samples do not have a chance to show in the graph. The feature is easily optimized in case the user wants to pause the plotting process in real-time, and the implementation for the visualization is a proof-of-concept, therefore it is not further discussed in detail in this subsection. Figure 5.11 presents a GUI for non-real-time visualization, in which sources of data can be retrieved by searching the database based on either sensor source id, or patient id. By clicking on a source from the result list and applying it, the user can perform visualization process by interacting with play, pause, write annotation, select channels, save annotation components in the GUI.

```
1  v.post(new Runnable() {
2      @Override
3      public void run() {
4          for(BitalinoDataSample sample: samples){
5              LineGraphSeries<DataPoint> tmp = channelLines.
                   get(sample.getChannel_nr());
6              if(tmp != null && isReady)
7                  tmp.appendData(new DataPoint(channels.get(
8                  sample.getChannel_nr()).getLastXRealtime()
                       ,sample.getSample_data()),
9                  true,NR_ENTRIES_WINDOW);
10         }
11     }
12 });
```

Listing 5.10: Update samples to GUI

Chapter 6

Evaluation

In this chapter, experiments are performed to determine whether the propositions of the database model and the database application design in Chapter 4 and Chapter 5 meet the user requirements. The experiments are real-time data collection and an overnight data collection, importing data from EDF file format, exporting data from the database to EDF file format, and querying data from the database. When performing a database evaluation, the most important metrics to evaluate are the read/write speed and the size of the database, because the time to retrieve data and space to store data must meet the user requirements. Moreover, the database application must ensure that no data is lost or data corrupted when storing and retrieving data. With respect to mobile platforms, which are usually limited resources and battery driven, the database application must also use the resources in an efficient way. The extensibility for future usage is clearly presented in the design chapter, and therefore not be evaluated in this section. As discussed in the design chapter, the database design is platform independent, it can easily be implemented on a bio-signals collector application such as CESAR acquisition tool to minimize the sending and receiving overhead between collectors and the database application server. There are some common experiments when evaluating a database design and database application. However, the experiments in this chapter are chosen to prove that the database model and database application are good candidates to be used to store OSA data on a mobile device. Therefore, experiments on storing real time data, overnight experiment, import and export to EDF files are performed in this section. In addition, a stress test experiment is performed to find the limited of the database application with respect to the resource usage and the number of channels it can manage.

The goals of the experiments are to evaluate the feasibility of the database model, and to convince the reader that the functional and non-functional requirements for the database modeling and the database application are satisfied by measuring performance of the read/write on the database, performance of the database application, and storage capacity. In this section, each experiment is described with respect to the reasons why it is performed, which workloads are used, which metrics are studied, the experiment itself, and the results from the experiment.

A BITalino plugged kit, two Android devices, and a CESAR acquisition simulator run on macOS Sierra are used for the experiments. Table 6.1 presents the specifications

Technology	Xiaomi Mi5	MacBook Retina 15 Late 2013	BITalino BT plugged kit
OS	Android 7.0	macOS Sierra	n/a
RAM	3GB	16GB	n/a
CHIP	Snapdragon 820 4 core 1.8GHz	Intel core i7 4 core 2.3GHz	MCU with sampling rate 1, 10, 100, or 1000Hz
Wi-Fi	802.11 a/b/g/n/ac	802.11 a/b/g/ac	n/a
Bluetooth	v4.2	v4.0	v2.0, range 10m
Battery	3000mAh	8440mAh	500mAh or 1500mAh

TABLE 6.1: Used devices specifications

of these devices.

6.1 Real-time data collection experiment

The database application is required to collect and store bio-signal data from the CESAR acquisition tool to the designed database model on a mobile device. The CESAR acquisition tool can send data with four different frequencies to the database, that are 1Hz, 10Hz, 100Hz, and 1000Hz. The goal of the database application is to collect OSA data. The application should be used for sleep monitoring to collect OSA data, which can last more than 8-10 hours. Therefore, We perform some experiments with a duration of more than 12 hours. To prove that the application properly works for a long time collection, an overnight experiment is presented in the next section. Currently, the CESAR acquisition tool cannot delivery samples at 1000Hz, it stops sending data after 30min-40min. Therefore, a CESAR acquisition simulator is used for sending samples with frequencies 1000Hz or higher. Goals of this experiment are to evaluate the database read/write performances and database size when collecting data from many different sources with different frequencies. Since the application is run on a mobile platform where the resources are limited, usage resources such as CPU and battery consumption are also evaluated in this section.

6.1.1 Experiment workloads

The database application collects samples from multiple sources simultaneously, in which each source sends data at different frequencies. The amount of samples the database needs to store at a certain time is depending on the number of currently connected sources and the number of channels that belong to each source. In addition, when the frequencies are high, the number of arriving samples in a time period becomes very large; which in turn has a huge impact on the storing process of the database application. Since the application uses batch processing to manage and

store incoming samples, the batch size strongly influences the performance of the application. The batch size is defined by multiplying the buffer duration with the arrival rate. Therefore, the workloads for the experiment are the buffer duration in second, the arrival rate of a source in Hz, and the number of channels (total channels from the connected sources).

6.1.2 Experiment metrics

Evaluation metrics must be meaningful to analyze whether the user requirements can be fulfilled. In the real-time data collection experiment, the application has to parse incoming data packets and transform them into useful objects for the database. The percent CPU usage and power consumption are considered the metrics used in the evaluation, because the application needs to create object for each incoming sample, then checking if the current batch must be sent to the thread that manages the database insertion. Percentage of CPU usage is obtained by using command "adb shell busybox top –d 5". This command generates an output which contains the information about CPU usage for each application, and the output is printed in the standard output in every five seconds. The outputs from the command are redirected to a text file, and the file is presented in charts. The python code for parsing CPU usage texts file into a chart is presented in Appendix C, in which Listing C.1 presents a sample output from the command, and a discussion about how to parse the output is also illustrated in the appendix. Power consumption can be obtained after a collection process is finished by using a built-in battery management application of the Android system (Settings - Battery - Battery use). Figure 6.1 presents an overnight experiment sample where the power consumption for each application is calculated in percent of the total power drained off.
Since the application uses batch processing to manage the incoming data, the SQLite transaction is requested for inserting data into the database when the temporary buffer is full. While the transaction is performed, it locks the entire database file. Hence, the SQLite time usage in millisecond for different batch sizes, and the number of insertions per millisecond are another metrics for the experiment. Since the application is implemented on the mobile platform where resources are limited, it is good to see how big the database grows in megabytes for different workloads in a specific period time.

6.1.3 Experiment setup

As described above, there are two workload generators that are used in the experiment, they are the CESAR acquisition tool, and the CESAR acquisition simulator. In addition, a control parameter which is the buffer duration for the application can be adjusted before performing a collecting process. A control parameter is a value that is adjusted before a collecting process starts is named the experimental workload. Calculated results after each collection process are named the responsive metrics in

the tables that are used in this section.
Table 6.2 presents the responsive metrics for an experiment where the experimental workload is a buffer duration; control workloads are arrival rate and number of used channels. In this experiment, arrival rate is 100Hz, and all channels of the CESAR acquisition tool are used (6 channels). Likewise, in Table 6.3, the experimental workload is arrival rate; control workloads are buffer duration (10 second), and all channels of the CESAR acquisition tool are used (6 channels). The number of channels is chosen as the experimental workload in Table 6.4. In this case, the control workloads are the buffer duration (10s) and arrival rate (100Hz).
For each experiment, a data collection process is iterated three times with a specific experimental workload value. That is, experimental workloads for the buffer duration (batch size) are 10 seconds, 20 seconds, and 30 seconds; experimental workloads for the arrival rate are 1Hz, 10Hz, 100Hz, and 1000Hz; experimental workloads for number of channels are 1 channels, 3 channels and 6 channels. The metrics that are evaluated for these iterations are the power consumption, SQLite usage time, and the size of the database after the iterations.
The duration for each iteration is about 60 minutes. Because the main metrics for the experiment are the SQLite write performance, resource usage, and the size of the collected data in the database, it does not need to perform a long time run for each iteration. Instead, only one overnight experiment is performed to demonstrate that the database application is to store all data from a long time run. The Xiaomi Mi5 phone is connected to the MacBook via a USB cable such that the percentage of CPU usage of the phone are sent to the MacBook. That is, an adb shell busybox command is run on the Terminal of the MacBook, and the percentage of CPU usage from the adb are redirected to a text file which is stored in the root of the folder project of the database application. The percentage of power consumption is obtained from the ”Battery use” of the Android operating system. However, the iteration needs to be repeated without connecting to the MacBook or a power adapter since the ”Battery use” does not perform the calculation unless the phone is powered by the battery. The database size in megabytes is obtained by getting the length of the database file, then dividing for 1024*1024 (converting from byte to megabyte). Listing 6.1 presents how to get the SQLite database size in the Android device. To calculate SQLite time usage, a timer is set before each SQL insert transaction. That is, the timer is started and stopped for each SQLite query execution, then the total SQLite time usage is obtained by adding all results from the executions.

```
1 File f = getApplicationContext().getDatabasePath(OSADBHelper.
      DATABASE_NAME);
2 long dbSize = f.length()/(1024*1024);
3 Toast.makeText(getApplicationContext(),"Database size "+String.valueOf(
      dbSize)
4                 +" MB",Toast.LENGTH_SHORT).show();
```

Listing 6.1: Get the size of a SQLite database in the Android

Responsive metrics	Experimental workload in 60 min Buffer duration		
	10 seconds	20 seconds	30 seconds
CPU usage (percent with 5s segment)	RTbd1.txt	RTbd2.txt	RTbd3.txt
Power consumption (percent)	2.5	2.6	2.4
SQLite time usage (millisecond)	194 242	193 130	184 811
Duration (millisecond)	3 668 780	3 750 028	3 711 576
Database size (Megabytes)	90	91	91

Table 6.2: Buffer duration

Responsive metrics	Experimental workload in 60 min Arrival rate			
	1 Hz	10 Hz	100 Hz	700 Hz (simulator)
CPU usage (percent with 5s segment)	RTar1.txt	RTar2.txt	RTar3.txt	RTar4.txt
Power consumption (percent)	<1	<1	2.5	2.7
SQLite time usage (millisecond)	14 311	39 442	194 242	2 012 042
Duration (millisecond)	3 658 002	3 679 001	3 668 780	3 737 474
Database size (Megabytes)	< 1	9	90	653

Table 6.3: Arrival rate

Responsive metrics	Experimental workload in 60 min Number of channels		
	1	3	6
CPU usage (percent with 5s segment)	RTnrc1.txt	RTnrc3.txt	RTnrc6.txt
Power consumption (percent)	1.5	2.2	2.5
SQLite time usage (millisecond)	54 613	113 050	194 242
Duration (millisecond)	3 678 704	3 675 105	3 668 780
Database size (Megabytes)	15	44	90

Table 6.4: Number of channels

6.1.4 Results and discussion

As presented in Table 6.2, 6.3, and 6.4, the power consumption is about 2.5% with arrival rates smaller than 100Hz. That is, the database application does not consume much battery, because the CPU time used to parse the arrival samples is low. The experiments, where the buffer duration and the number of channels are used as experimental workloads, illustrates that the power consumption and the CPU usage have a small variation. It is because the arrival rate for these experiments is the same. The variation occurs in these experiments since different buffer durations and number of channels causes different overhead when inserting samples to the database. Therefore, the power consumption and CPU usage are mainly depending on the arrival rate, the higher the arrival rates, the more time the CPU time it takes to parse the arrival samples, hence the more power is drained.
The write performance of the database application does not depend on the arrival rate and the number of channels, it is mainly depending on the size of the buffer (the batch) that groups all arrival samples into a insert transaction. In theory, the more samples in groups, the less time is needed for writing to the database. However, the performance of the database application is not increased so much. As presented in Table 6.2, tripled the buffer duration, the performance is increased (184811*100%)/194242 which is about 9.5%. Therefore, a fixed duration of the buffer is recommended to use for the database application on the Android platform.
As presented in Table 6.3, and 6.4, the size of the database is depending on the arrival rate and the number of channels that are used to collect samples. If sensor kits deliver data tuples with a frequency of 1Hz, the database can continuously store data for six channels up to 41 days by using 1GB memory, that is, one megabyte per hour which is 1000 hours for 1GB or about 41 days. Hence, the Xiaomi phone that is used in the thesis can continuously collect and keep data for 20GB * 41 days = 820 days, or 2,2 years without backing up to a stationary computer. However, high frequencies offer better data for later analysis, hence, 10Hz, 100Hz and 250Hz are often used to collect bio-physiological signals. The size of the database is linearly increasing when the frequency is increased. With frequencies from 100Hz to 700Hz, the collected data needs to be backed up to a stationary computer after one or two overnight collections.

6.2 Overnight experiment

An overnight experiment is performed in this thesis to demonstrate that the database application can collect data for a long period period of time. In addition, the feasibility of the database model and database application for monitoring sleep during a whole night is also analyzed by assessing metrics such as the power usage, the size of the collected data, time that the database application accesses the database file, etc. In this experiment, the database application receives data from the CESAR acquisition tool until the tool or a users disconnects the connection.

6.2.1 Experiment workloads and metrics

Workloads that are used in this experiment, are all channels from the BITalino sensor kit (6 channels). Each channel collects data with 100Hz frequency, which is 100 samples per second. Therefore, the number of samples the database application needs to put to the database in one second is 600 samples. To avoid overhead when accessing the database file for storing samples, the database application uses batch processing. The batch size that is used in this experiment is 5 seconds. That is, the application groups 5*600 = 3000 samples into one insert transaction (instead of 3000 transactions if the batch processing is not used). The metrics used in for this experiment are the percentage of power consumption of the total power usage of the mobile device, size of the collected data in megabytes, number of samples in the database, number of received packets, percent time accessing the database file over the experiment duration.

6.2.2 Experiment setup

The BITalino can be powered by different battery capacities; a 500mAh and 1500mAh, which can offer about 6 to 20 hours for collecting data with 100Hz each channel. However, only the 1500mAh battery is used in this experiment, because early tests showed that the 500mAh battery can not be used for performing the overnight experiment. The 1500mAh battery of the BITalino sensor kit is full charged before the experiment. In addition, the Xiaomi Mi5 phone is plugged to a power adapter, because the current version of the CESAR acquisition tool needs the display of the phone to be on to avoid to be killed by the Android system because it is running for a long time. This configuration does not impact the experiment, since the metric that is used for evaluating the power usage, is the percentage of power the application consumed over the total used power during the experiment. The phone is full charged and disconnected from the power adapter before the experiment, then the percentage of power consumption of the database application is gotten from the built-in battery management application of the Android operating system when the battery power is about 50%. In other words, the percentage of power usages for applications, which are running under the experiment, are stable, therefore, the power usage of the database application can be evaluated without waiting for the battery of the phone is run out. After obtaining the percentage of the power usage, the phone is connected to the power adapter to continue the collection process. The total SQLite time usage is calculated by adding all timed insertions of each transaction from the batch processing. The duration is calculated by subtracting the maximum timestamp and the minimum timestamp of the samples in the overnight record. The percentage of power consumption is gotten from the built-in battery management application, which is named ”Battery use”, of the Android operating system. The number of packets can be gotten from either the CESAR acquisition tool, or counting number of samples in the database dividing to number of channels that are used in the experiment (6 channels). To get the number

Start time	20.04.2017 17:43:55
End time	21.01.2017 12:32:20
Duration	67 705 166ms = about 18 hours, 48min
SQLite time usage (millisecond)	4 251 412 ms
Percent SQLite time usage over the duration	4 251 412/ 67 705 166 = about 6.3%
Power consumption (percent)	6.6% of total power usage
Number packet from the CESAR acquisition tool	6 770 896 packets
Number of samples	40 625 376
Database size (Megabytes)	1736 MB

TABLE 6.5: Overnight experiment

of collected samples, a simple SQL query is performed, i.e., "SELECT COUNT(*) FROM SAMPLE". Since the database is empty before the experiment is performed, the database size is also the size of all data collected after the experiment performed. The results from the experiment are presented in Table 6.5.

6.2.3 Results and discussion

For the write performance of the database application, as presented in Table 6.5, the number of samples are that collected during the experiment is 40625376, and the total time used for the insert transactions is 4251412 millisecond. Based on these numbers, the average time used for each transaction is 4251412/(40625376/3000) = 314 millisecond, where 40625376/3000 = 13542 is the total number of insert transactions performed. That is, the database uses 314 millisecond to store samples from 5000 millisecond (5 second buffer). Hence, the percent time usage for accessing the database file (I/O time usage) is (314/5000)*100% which is about 6.3% of the idle, it can also be calculated by using the total SQLite time usage and the duration as presented in Table 6.5. Based on the percent time usage, the database application can manage up to 15 BITalino sensor kits where all channels of the kits are used, and the I/O time usage to access the database file would be 15*6.3% = 94,5% idle. 15 BITalino sensor kits offer 15*6 = 90 different sensors that can be used to collect data at 100Hz, or 15 users can use one Xiaomi MI5 phone to store data simultaneously.

For the power consumption, the database application does not use much battery compared to other applications. Figure 6.1 presents the power consumption for applications that are running during the experiment. The CESAR acquisition tool consumes about 31.1% while the database application consumes 6.6% of total power usage. Hence, the application is considered quite efficient in the case of power usage. The overnight experiment is performed during 18 hours and 48 minutes, which is

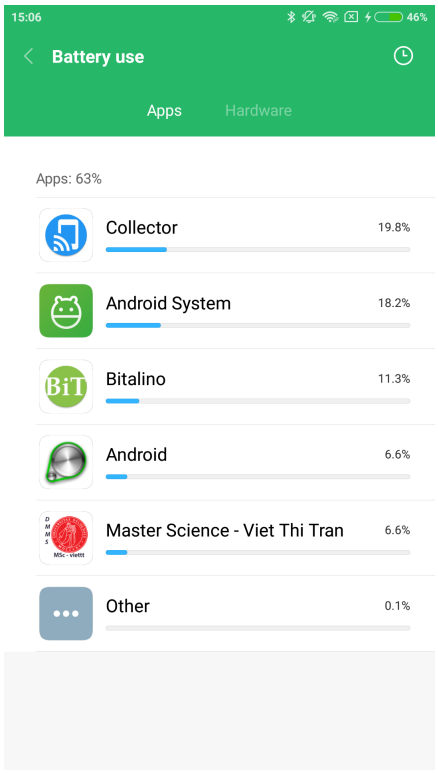


FIGURE 6.1: Battery usage under overnight experiment

longer than a normal nighttime sleep. The data size after collecting data for nearly 19 hours is 1736 megabytes (1.7 GB). Currently, an Android phone usually has 16 to 64 gigabytes(GB) internal memory, in addition, an external Secure Digital (SD) card with 16GB costs about 80kr. With a 16GB SD card, a user can store up to nine 18 hours (overnight) records, or nine users can use the same SD card to store their collected data. The phone that is used in the experiment has 32GB memory which offers 15 users use it simultaneously. It is 15 users, because the write performance from the previous discussion can handle up to 15 users.
Nonetheless, the results from the experiment present that the database application is properly good to be used for a long period time of data collection. Moreover, multiple users can share an Android mobile device to store their data. Furthermore, the power consumption and data size are properly good enough when collecting data for a long period of time.

6.3 EDF import

The database model is designed in a way that opens for all bio-signals, and can store data from other database sources if the sources support EDF exporting. In other words, the data base application is required to import data from EDF and EDF+ file formats. Goals of the experiment are to evaluate database writing performance and usage resources such as CPU and battery consumption, especially, when the application reads files simultaneously.

6.3.1 Experiment workloads

The database application reads data samples from multiple files simultaneously. Ideally, each sample from the files is parsed to an object in memory, then inserted into the database. However, each INSERT statement in SQLite has its own transaction, and SQLite can only do a few dozen transactions per second. The samples are, therefore, grouped into a transaction before performing a SQLite insertion. The transaction locks the entire database file when it inserts the data. Hence, if the sample group is too large, it can cause either a memory problem, or poor performance since other transactions have to wait for it. Therefore, the workloads for the experiment are the size of the sample group and the number of file reads.

6.3.2 Experiment metrics

Metrics that are chosen to highlight the goals and workload of the experiment are the percentage of CPU usage, the percentage of power consumption, the size of a EDF file in the database, and the total time the experiment holds the shared SQLite connection. The percentage of CPU usage and the power consumption need to be evaluated since the application needs to parse the samples in the EDF file format to objects that can be stored into the database. In addition, the time that is used to insert samples can be used for evaluating the writing performance of the database.

6.3.3 Experiment setup

```
1 (no options)Terminal $mit2edf -r a01
2 (options)Terminal $mit2edf -r a01 [-v -h -p] -o phy1.edf
```

Listing 6.2: Convert mit2edf by using terminal

To generate workloads for the experiment, a function from Physionet which is named mit2edf is used for exporting bio-signal data from Physionet databases to the EDF file format. As presented in the manual page of the function, mit2edf creates an EDF file that contains the same data as the input files which are in form of Waveform Database format (header and signal files). However, the function is included in the WFDB software package from Physionet, and the user must install the package first. There are three options when using the function which are printing a brief usage summary, choosing a name for the exported edf (the exported file has the same name as the imported record as default), and printing debugging output. Listing 6.2 presents how to use the function to convert a01 record to EDF file format without options and with options on a Linux/Unix shell. Workloads that are used for the experiment are a01.edf (5,64 MB), a02.edf (6,07 MB), and a03.edf (5,98 MB) files, which are generated from a01, a02, and a03 records from CinC2000 data sets. Since the implementation for the EDF-reader uses the same algorithm (batch processing) as real-time importing, there are two factors that influence the goals of the experiments. These are the number of simultaneously EDF files read, and the buffer size (i.e.,

Responsive metrics	Experimental workload in 60 min Number of files simultaneously read		
	1 (5,64 MB)	3 (11,71 MB)	6 (17,69 MB)
CPU usage (percent)	edfFI1.txt	edfFI2.txt	edfFI3.txt
Power consumption (percent)	2.4	4.8	6.6
SQLite time usage (millisecond)	228 235	466 203	703 837
Database size (Megabytes)	100	218	335

TABLE 6.6: Number of files simultaneously read

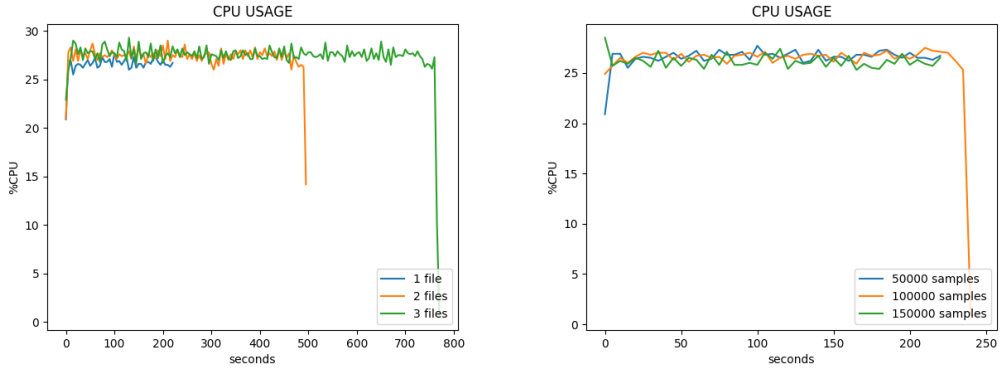
Responsive metrics (a01.edf-5,64MB)	Experimental workload in 60 min Number of samples in batch		
	50 000	100 000	150 000
CPU usage (percent)	edfFIS1.txt	edfFIS2.txt	edfFIS3.txt
Power consumption (percent)	2.4	2.0	1.7
SQLite time usage (millisecond)	228 235	209 936	209 442

TABLE 6.7: Number of samples in batch

batch size). Table 6.6 presents the results for the experiment where the experimental workload is the number of simultaneously EDF files read. In this experiment, the batch size is 50000 samples, and is kept the same for three iterations. Table 6.7 presents the results for the experiment after three iterations with batch size 50000, 100000 and 150000 samples respectively. Since the number of file read is not the experimental workload in each iteration, the number of file read and the used files must be the same for each iteration. a01.edf is chosen as the control workload for the experiment. Figure 6.2 presents the CPU usage for each workload.

6.3.4 Results and discussion

When increasing the number of files that are simultaneously read, the responsive metrics are linearly increasing. Although each input file is managed by a separate thread, it must wait for accessing and writing to the database. The larger the number of file read, the longer the time to write to the database file, hence, the power consumption is increased. As presented in Figure 6.2(a), the CPU usage is increasing when the number of file reads is increased, but it is quiet stable between 25% to 30%. This is because the waiting time for I/O is much longer than the CPU time. In contrast, when increasing the batch size, the power consumption is decreased, because the database application reduces the overhead for writing data to the database file. However, the CPU usage is likely the same for each iteration, because the CPU time



(a) Number of file *(b)* Buffer size

FIGURE 6.2: CPU usage for the EDF import experiment

that is used to parse samples in the a01.edf file is mostly unchanged.
As presented in Table 6.6, the EDF files are 18 times bigger when they are stored in the database file, because the EDF/EDF file formats do not include metadata in a sample while the database model includes metadata such as timestamp and the record id a sample belongs to. Moreover, each sample in a EDF file i limited to 2 bytes integer while a sample in the database is 8 bytes float. However, the metadata used in the database model provide many advantages for working with data, such as leveraging the advantages of SQL, easy to mixed multiple channels from different sources, etc.
The a01.edf file contains 2958000 samples, and the database application took 228235 milliseconds to insert the samples to the database. It is about 12960 samples per second. The time to read a EDF format file is therefore depending on the number of samples the file contains. The time for the reading process is decreasing when the number of samples in a EDF format file is increasing.
The resource usage is efficient, but the performance is poor when importing multiple EDF/EDF plus format files simultaneously. This is because the files take turn to write their data to the database file. Parallel reading files does not increase the database writing performance, but it increases the performance for parsing data from EDF files.

6.4 EDF export

The database application is required to export data to EDF file format for sharing or for backup data purposes. Since each record in the database is defined for a specific channel of a source (a.k.a channel record), and if records have the same patient id, physician id and timestamp, they are considered to be belonged to the same collecting process (a whole record). It is because a patient can use multiple sensor platforms to collect data. By choosing records with the same patient id, physician id and timestamp, a whole record can be retrieved. A user can choose which channel records to

be included in the exported record, because sometimes the user needs data from some specific channels. The EDF file format requires that a whole record needs to be divided into small data records. The duration of the small data records is freely chosen. However, a data record size does not exceed 61440 bytes (edfplus.info, n.d.[c]). In the implementation of the database application, a data record is defined as a buffer in memory, and this buffer is flushed to the exported file when it is full. Hence, the data record size is also the buffer size which is defined by a summary of all channel buffer sizes. A channel buffer size is defined by multiplying the data record duration with the frequency of a channel.

$$bufferSize = \sum_{i=1}^{n} \frac{duration}{frequency\ channel(i)}$$

Goals of the experiment are to evaluate the read performance of the database and resource usage such as percent CPU usage and battery consumption.

6.4.1 Experiment workloads

As presented in the design of the database application, the application continuously queries parts of a record into a buffer and writes the full buffer to the exported EDF file. The application requests database access for each selected record. If the number of selected records for exporting is large, the application needs to access the database multiple times. Therefore, the experimental workload for the experiment is the number of records to be exported.

6.4.2 Experiment metrics

A record can be very large, since it can be collected in long time, and it takes more time to export the data. The database application needs to query data multiple times to avoid memory corruption since the record cannot be kept in memory for exporting. Therefore, metrics such as CPU usage, power consumption, database read performance, and shared SQLite connect time usage are chosen as the metrics for the experiment.

6.4.3 Experiment setup

The records in the database are collected from the BITalino sensor kit during a period of 60 minutes, and all channels of the BITalino are used. Each channel delivers samples with frequency 100Hz. Table 6.8 presents the results for the experiment where the experimental workload is the number of exporting record. The experiment is performed with 1, 3, and 6 records for each iteration. The data record duration that is used in each iteration is five seconds. In this table, the size of collected data for each channel is equal, hence, data size in bytes for each experimental workload is calculated based on the total size of the database and number of exported channels.

Responsive metrics	Experimental workload in 60 min Number of exporting records		
	1 (15MB)	3 (45MB)	6 (90MB)
CPU usage (percent)	edfFER1.txt	edfFER2.txt	edfFER3.txt
SQLite usage time (millisecond)	28 180	83 271	165 855
Exported file size (Megabytes)	0.727	2.13	4.26

TABLE 6.8: Number of exporting records

That is, data of six channels are stored in the database, each channel stores 15MB of data in the database.

6.4.4 Results and discussion

As presented in Table 6.8, the exported files are about 22 times smaller in size compared to the size they occupy in the database. It is very efficient when the database needs to back up the data to a remote storage server, but it takes about 166 seconds to query 90MB data, or about 0,5MB per second, which is quite slow. However, the database application queries a part of the data into a buffer before writing to the exported file, the procedure is repeated multiple times until the entire record is exported. This action causes more overhead if the buffer is small. The percent CPU usage for this experiment is not so high and stable at 26% to 27%.

6.5 Querying data experiment

When processing a SQL query, a database management system (DBMS) goes through at least three steps. The first step is to parse the SQL query into a data structure that can be processed by the DBMS. In this step, the syntax and semantic of the query are checked to determine if the SQL statement is syntactic validity, and if the statement is meaningful, i.e., whether the columns and tables are existing in the database. The second step is to optimize the query. That is, by using a set of equivalence rules, the query optimizer can generate many equivalent plans in which a cost is assigned for each plan. The plan that has the lowest cost is taken to the third step which is to execute the query plan. Although the SQL query is optimized before querying, the complexity of the SQL query and the results from the query are not changed.
Goals of the experiment are to evaluate the resource usage and performance of the database application with respect to the complexity of the queries, and the size of the results.

Table	Number of columns	Approximate records
SensorSource	3	<100
Patient	7	<100
Physician	4	<100
Clinic	5	<100
Person	8	<100
Channel	11	<1000
Record	10	<1000
RecordAnnotation	2	<1000
Annotation	5	<1000
Sample	3	>1 million

TABLE 6.9: Datasets in the database

6.5.1 Experiment workloads

In SQLite, all queries are simple select statements. A simple select statement is often processed via four steps (SQLite, 2017a).

- From clause processing: The input data of the statement are specified in this step.
- Where clause processing: The expression in the clause helps to filter the return data.
- Group by, having processing: groups of data are calculating and aggregating with respect to the filter expression in having.
- Distinct / ALL keyword processing: result rows are return (duplicate rows are removed if "distinct" is specified).

A compound query statement is defined via connecting multiple simple select statement by using UNION, UNION ALL, etc.
Workloads for the experiment are therefore depended on the FROM clause, and whether statements are compound statements. As presented in SQLite (SQLite, 2017a), all join-operators are based on Cartesian product of the left hand side (N rows) and right hand side (M rows) of the datasets. The results from the join are therefore NxM rows. In other words, the complexity when joining two table is O(MxN). The complexity is increased by the number of tables that are stated in the FROM clause. Moreover, if the result from the query is too large to hold in the memory, the DBMS musts pass the database multiple times (multiple-passes).

As presented in Table 6.9, the heaviest table is the Sample table, and together with the complexity of queries, they are therefore considered the main workloads of the experiment. There are four queries that are used for evaluating the experiment; the queries are described as follow:

- **Query 1:** A relatively small dataset is scanned (a.k.a simple query with small result). Any tables presented in Table 6.9, as long as not the table Sample, can be chosen for this query.

Query nr	Table(s)	Nr record input	Nr record output
1. Simple query-small result	Channel	6	6
2. Simple query-large result	Sample	2 231 580	2 231 580
3. Complex query-small result	All tables	160 673 760	1
4. Complex query-large result	All tables	160 673 760	160 673 760

TABLE 6.10: Statistic for the queries

- **Query 2:** A relatively large dataset is scanned (a.k.a simple query with large result).
- **Query 3:** All presented tables in Table 6.9 are joined, then an aggregate function, i.e. count, sum, max, etc., is applied to have a small result (a.k.a complex query with small result).
- **Query 4:** All presented tables in Table 6.9 are joined, no filters are applied in this query (a.k.a complex query with large result).

6.5.2 Experiment metrics

Metrics that are used for evaluating the experiment must highlight the resource usage and performance of the database application when executing different queries with different complexity. Therefore, the metrics that are chosen for the experiment are percent CPU usage and executing time in millisecond for queries.

6.5.3 Experiment setup

The described queries in the subsection workload of this experiment are translated into SQLite code as follow:

- **Query 1:** The whole table Channel is chosen to be scanned.
  ```
  select * from Channel
  ```
- **Query 2:** Since the table Sample is the largest, it is chosen for this query.
  ```
  select * from Sample
  ```
- **Query 3:** This query counts number of rows return from Cartesian product all tables.
  ```
  select count(*) from SensorSource, Patient, Physician, Clinic, Person,
  Channel, Record, RecordAnnotation, Annotation, Sample
  ```
- **Query 4:** This query return all rows from Cartesian product all tables.
  ```
  select * from SensorSource, Patient, Physician, Clinic, Person, Channel,
  Record, RecordAnnotation, Annotation, Sample
  ```

Table 6.10 presents query statistics after executing the queries, while Table 6.11 presents the results for the experiment. The records in the database are collected from the BITalino sensor kit during a period of 60 minutes, and all channels of the BITalino are used. Each channel delivers samples with frequency 100Hz.

Responsive metrics	Experimental workload in 60 min			
	Simple query small result	Simple query large result	Complex query small result	Complex query large result
CPU usage (percent)	Qr1.txt	Qr1.txt	Qr1.txt	Qr1.txt
Executing time (millisecond)	1	1 612	8 958	509 727

TABLE 6.11: Executing time and resource usage for the queries

6.5.4 Results and discussion

As presented in Table 6.10 and 6.11, it takes one millisecond to scan a small dataset, which is 6 rows from the table Channel in this case. Since the dataset is very small, the main purpose of the first query in the Table 6.10 is to find the overhead when the database application accesses the database. To find the reading speed of the database application, a large dataset must be used, and it can be done by either scanning the entire table Sample, or Cartesian product table Sample with other tables as long as the tables have at least one row. It takes 1612 milliseconds to scan an return 2231580 rows of the table Sample; the reading speed (with all rows returned) is therefore 2231580/1612 = 1384.354 rows per millisecond or 1384354 rows per second. Likewise, the reading (with all rows returned) speed for query number four is 160673760/509727 = 315.251 or 315251 rows per second. The reading speed is decreased when increasing the number of returned rows, because when the returned data are large such that they cannot be kept in memory with one database accession, the database management system needs to access the database file multiple times. As presented in Table 6.11, the query number three returns only one row, which means that the database management system does not need to perform many accesses to the database file. The overhead for multiple accesses is therefore removed, and the reading speed is 17936.342 rows per millisecond, or 17936342 rows per second. It is about 17936342/315251 = 56 times faster than the fourth query. Hence, with respect to the hardware platform the reading speed is mainly depending on the memory of the devices. With respect to users, algorithms that are used to get data have a huge impact on the performance of the database application. How to create a good query plan, and how to optimize SQL queries is a wide topic and therefore not to be evaluated in this section. The results from the tables show that it is quite fast to get all samples of a 60 minutes record, and the reading time is linearly increasing when the number row read is increased. Although the hardware of the mobile devices has improved, the mobile devices still can not replace stationary computers. However, these results shows that some simple queries can be performed on the mobile devices, or even some mining methods can be applied on the database before the data are sent to the stationary computer. For example, to scan and filter samples in a 60 minutes record is quiet fast (about 1.6 second) on the used mobile phone, so prefiltering data on a mobile device before sending to a stationary computer is good to be considered.

The percentage of CPU usage is about 3% to 7% for the first and the second query, but it is about three to four times higher (about 28%) when executing the third and the fourth query. The resource usages are therefore acceptable.
In conclusion, if the running time for mining tasks is not critical, the database application can create threads to manage the mining tasks. The results are stored to file or sent to the physician whenever the tasks are finished. Otherwise, the raw collected data should be extracted and sent to the physician. As the results presented, time to extract data from the database is fast,i.e., about two second to get a 60 minutes record which contains six channels with 100Hz. Thus, the relation database is good to be used to store the OSA data.

6.6 Stress test with visualization and mixed tasks experiment

It is possible that a user collects data from multiple sensor sources to the database, and plots the collected data on a graph while there is an exporting process is running. In other words, the application usually performs multiple tasks simultaneously rather than finishes a task before beginning the others.
Therefore, the goals of this experiment are to evaluate the robustness of the database when collecting data from many different sources (both EDF files and CESAR acquisition tool), and resource usage such as CPU and battery consumption.

6.6.1 Experiment setup

Each experiment, which is presented the previous sections, is evaluated with different workloads. These workloads are increased after a certain unit of time until the application is stopped or crashed. That is, the BITalino sensor kit is connected to the database application to send its data, then multiple simulators of the CESAR acquisition tool are created and connected to the database application. While the sensor sources send their data, the a01.edf file is chosen to import into the database. Right after the file beginning to load, data that are stored from the BITalino sensor kit are chosen to export to a EDF file. After that, a random source is chosen to plot data to the graph. The tasks to connect a source, import EDF files, and export to EDF files are repeatedly performed. The experiment terminates when the application is not responding or it crashes.

6.6.2 Results and discussion

As presented in Figure B.5, the CPU usage is about 10% with a single sensor source which has six channels and each channel sends data with 100Hz. When the second source (100Hz), the third (600 Hz) and the fourth (600 Hz) are added, the CPU usage is about 26%, 34%, and 45% respectively. With importing and exporting edf files, the CPU usage reaches nearly 60%. As presented in the figure, the CPU usage

suddenly drops to 20% after the edf files are loaded and exported. It is because the garbage collector is invoked to reclaim the usage memory. That is, when there is not enough memory space for the application, the Android operating system suspends all threads, applications with low priority (i.e., do not run in the foreground or do not have any services) are killed to get more space for the database application. With visualization, the CPU usage is about 50% while the application collects data from the sources. The speed of incoming data and the writing speed of the database application are balanced. After that, another source is connected, the a02.edf file is imported and six random channels are selected to export. As presented in B.5, the period from the second 1500th to 2000th, the percentage of CPU usage begins to behave strange after performing these action. The reason for this behavior is the amount of data that is waiting for read/write data from/to the database, hence, these data need to be kept in memory. Because the buffer grows quite fast, the Android operating system needs to suspend all threads to claim memory. However, the Android operating system can not claim the memory usage from the buffer that is used by the database application, because the database application is running in the foreground and has services to collect data from sensor sources. Therefore, the reclaim process is repeated until the Android operating system decides to kill the application.

As discussed in the previous experiments, the database application can manage up to 15 BITalino sensor sources (100Hz) if only data collections are performed. For mixed tasks, the database application can manage four BITalino sensor sources, in which two sources send data with 100Hz, while the last two send data with 600Hz, which is equivalent with 14 BITalino sensor sources (100 Hz). The database application is killed by the operating system when we attempt to add one more source while importing and exporting data.

Data visualization has a small impact to the database application, because the CPU usage is lower than 60% regardless plotting data to the graph or not. The performance of the database application is mainly depended on the read/write speed of the chosen database management system, SQLite in this case. Some parts of the implementation do not use background threads to get data from the database, which causes freezing the GUI if the query to get data has to wait for the other queries. For example, when users create a new record, queries to get the patient, clinic, and physician information are performed such that the users can select these information from the returned list. Moreover, in the raw query part, it must be good to prevent the users from submitting the same query if they misunderstand that the query they have submitted is not executed, because some query may take long time. Nonetheless, the implementation is a proof of concept and therefore does not take the user experiment (UX) into consideration.

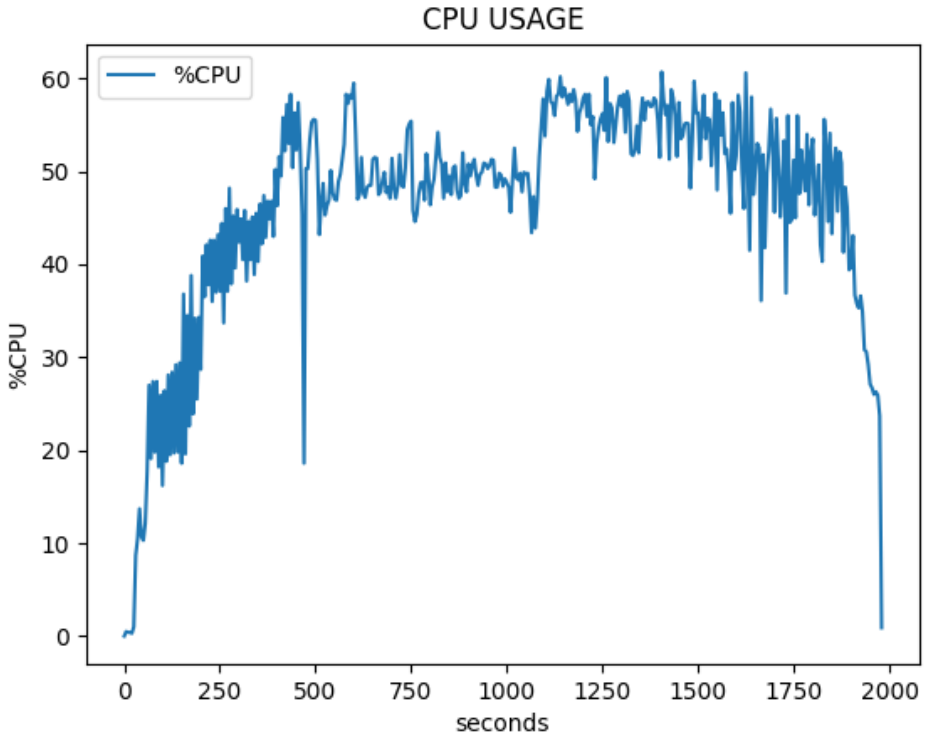


FIGURE 6.3: CPU usage under stress test

6.7 Summary of results

The main goals of this chapter are to convince the readers that the database design and database application meet the requirements to store OSA data, and to illustrate that a relation database solution on a mobile platform, which is used to store not only OSA data, but also other physiological data, is worth to be considered. Thus, the key results from the experiments in this chapter are presented and discussed to show that the database design and database application on a mobile platform are feasible to store OSA data, and extensible for other physiological data.

The database application is required to parse the incoming samples from sensor sources to tuples and store them into the database. As presented in the real time data collection experiment, the database application can continuously collect and keep data for more than 2 years for six channels with a frequency of 1Hz, or a whole day with a frequency from 100Hz to 700Hz on 20GB storage space. The percentage of power consumption is about 2.5% which is quite efficient to use for a long time running. The overnight experiment, which uses the BITlino sensor kit where six channels with a frequency of 100Hz are used, shows that the database application consumes about 6.6% of the total power usage, and the size of the database after collecting is 1.7GB. This means that the mobile device can store up to nine overnight records with this configuration on a 16GB SD card.

In case a user want to send the collected data to the physician, the collected data are

exported to a EDF file which is designed to minimize the data size for sending and sharing data. The physician can import the EDF to a mobile which is implemented the database application, or can use other database analysis tools to analyze the received data from the EDF file. As presented in the import and export experiments, the data that are stored in the EDF file is about 18 times smaller as they are stored in the database, which is very efficient for sharing.
In some scenarios, i.e., a patient snores for a long period of time or airflow from the nose is dramatically reduced, to analyze the collected data directly on the mobile device and send the results to a stationary computer is better than just simply sending them to the stationary computer. By leveraging SQL, some mining task can be performed on the collected data, then only the results are sent to the stationary computer. As presented in the querying experiment, the query is executed fast if only few results are returned. To apply a mining task where a small set of rows is return, i.e., to find periods that are bigger than 30 seconds in which a patient snores, is good to be considered.
Last but not least, the stress test presents that the database application can collect data up to 84 channels (14 sensor kits, each sensor kit uses six channels to deliver data) with a frequency of 100Hz, which is possible to collect data from most of the positions on a users body and some data from environment around the user, i.e., the light, sound, etc. The collected data can also be visualized on a graph which offers users a better understand on their data.

Chapter 7

Conclusion

In this chapter, the main goals of the thesis with respect to the problem statement stated in Chapter 1 and the ways the thesis is done are summarized in Section 7.1. Section 7.2 presents the works that are related to this thesis. In addition, how the thesis contributes to overcome the research problems is also presented in this section. Problems that have not been solved in the thesis are presented in Section 7.3, then some suggestions for further research are discussed in the last section.

7.1 General summarization

The main goals of this thesis are to design a relational database model, and to develop a database application to store Obstructive Sleep Apnea (OSA) data and that is extensible for other bio-physiological data. To pursue the goals, the research problems are stated, the works that this thesis is based on are discussed, then the pros and cons of the related works on the problems are presented.
At first, a literature overview of sleep apnea and its' characteristics are presented in Chapter 2. It is because the designed database model and database application are mainly used for storing and analyzing signals from this illness. Since the thesis uses the CESAR acquisition tool to collect data and EDF/EDF plus file format to backup data, their characteristics, i.e., collected sources, platform used, formats for transferring and storing data, supported functions and documents, etc. are carefully studied in Chapter 3. In addition, the requirements for both the database model and database application are analyzed. For database modeling, entities are derived from the observable signals and characteristics of OSA, and relationships between the entities are defined. The database modeling process is performed, which results the logical database model used in the thesis. For the database application, two wrappers are provided for importing data from real time sources and non-real time sources, then six experiments are performed to prove that the thesis has solved the research problems which are stated in Section Problem statement in Chapter 1. The main focuses of the experiments are the feasibility of the database design and the performance of the database application on a mobile device powered by Android. That is, the database application can perform multitasking, and can collect data from the CESAR acquisition tool for a whole night without losing data. At first,

each function of the database application is evaluated separately, then stress tests are performed where the functions are simultaneously evaluated with high workloads.

7.2 Related works

OSA data need to be stored and retrieved for analysis purposes in an efficient way. There are many efforts to collect and store the OSA data, however each clinic may determine which hardware platform to buy, therefore, the clinic defines its own methods to collect and manage the collected data based on the chosen hardware. Most of clinics do not share the ways they manage the collected data, i.e., the Physionet databases, Nox Medical, but they often offer a software package to export their data to a common file format, i.e, EDF file format. Currently, to store OSA signals, most of clinics choose to use files to store the collected data. For example, the Physionet databases use multiple files and file formats to keep a record, i.e., the record a01 that is used in the EDF import experiment is kept in 12 files in the Physionet databases. Likewise, the EDF file format is chosen to use to store the OSA data in the DREAMS Apnea Database (Mons Universite, retrieved 2017). At the time of writing, the OSA data are only stored in files, and a relational database to store the data is not considered yet. However, relational databases and database management systems (DBMSs) are introduced to store other physiological sensor data, i.e., A SQL Database for Sensor Nodes and Embedded Applications (a.k.a LittleD) (Graeme, retrieved 2017), A Database in Every Sensor (a.k.a Antelope) (Nicolas, retrieved 2017). These two database are implemented on sensor devices where the resources are more constrained compared with a mobile phone. The users not only can store the sensor data into the relational database on the sensor devices, but also can analyze the collected data by either directly, or remotely executing SQL queries on the sensor devices. However, DBMSs on the sensor devices provide limited SQL functions. Moreover, to implement and use these DBMSs is difficult, i.e., the database design is mostly depended on these DBMSs. In addition, the DBMSs are not frequently updated compared to a DBMS on the mobile device or the stationary computer, i.e., SQLite or PostgreSQL. Therefore, in this thesis, a relational database model is designed to store OSA data on a mobile device with minimizing resource usages, i.e., low power consumption, storage space and memory used, CPU used, etc., and maximizing data usage, i.e., easy to extract, analyze, backup, restore, share, etc., is provided. The relational database model in this thesis is platform independent, and it can therefore be implemented on different mobile operating systems, i.e., iOS, Android, Windows Phone, etc., which are integrated a DBMS, i.e., SQLite. A database application on the Android operating system is also provided in this thesis to evaluate the feasibility of the database and database application on a mobile platform, i.e., the Android operating system.

7.3 Contributions

This thesis presents a solution for an open database model and a database application for storing OSA monitoring data and other bio-physiology signals on Android devices. The database application allows to store monitoring data from the BITalino sensor kits, and to import existing polysomnogram data from EDF/EDF plus files. To avoid reinventing the wheel, the thesis makes use of the advantages from the CESAR acquisition tool, which has high performance, low resource usages, and stable when collecting data for long time. The CESAR acquisition tool provides an interface to send packets via TCP/IP; the database application therefore provides a friendly user interface to manage the connection from the tool to the application. The application allows multiple sensor sources to connect and store their data to the database on the device simultaneously. A list of current sources is presented via a graphical view, in which the status of each source is illustrated whether the source is active, saving its data to the database, or shows its data on a graph. Each sensor source is managed by a separate thread, which allows maximizing performance when collecting data in real time.

For backing up and restoring data, the database application provides two functions which are EDF/EDF plus importer and EDF/EDF plus exporter. It is because the storage space requirements are minimized when using these formats. Moreover, when sending or sharing data, the bandwidth used is low compared to other file formats. A user can import a EDF file by choosing the file from mass storage such as SD card via a graphical file chooser. A process bar indicates how many percent of a file have been imported, and a user can query, share, analyze, etc., on the imported part of the file. It is not logic for other data types, but it might be a good option for the bio-physiological data, because the file is treated as a stream of data. If a user needs only some parts of the file, the user does not need to wait for the whole file to be loaded.

By providing a relational database solution for storing polysomnogram data, the thesis can make use of SQL language and SQL function to analyze data directly on a mobile device. Moreover, the database application also allows users to easily choose parts of recordings, combine data from multiple channels to do comparison such as the quality of collected data, or to export to a EDF file for sharing purpose. Combining data from multiple channels also allows a patient to use different sensor data kits to collect data into one record.

Nonetheless, the database model and database application are designed in the way that provides maximum reliability and user friendliness. That is, the application takes the user experiment into consideration, which means many users prefer to interact with symbols and icons compared to text. The users do not need to have a user manual to use the application. However, to provide the users a comprehensive information how to use the application, a user manual is provided in Appendix A. The reliability is proven via stress test in the last subsection of the evaluation chapter. That is, the

application can manage all channels of 14 BITalino sensor kits while importing and exporting data without crashing, or losing data.
Therefore, the database model and database application that are provided in the thesis are considered efficient for storing, managing, and analyzing OSA signals, and other bio-physiological signals.

7.4 Open problems

Considerations concerning the chosen platform
The database application is required to run for a whole night or at least 12 hours without interruption. That is, it must not be suspended, or not be killed by the operative system. It is because the database application collects data from sensor sources in real time. However, the API guides from the Android developers (Android, retrieved 2017[b]) illustrates that Android might decide to kill a process at some point. The main goal of this action is to try to keep a certain amount of memory (RAM) for smooth experience, and to save battery as there are less applications using the phone's CPU. Once the application is killed, all components running in it are consequently destroyed. Victims that seem to be killed by Android are usually applications hosting activities that are not visible on screen. Currently, the CESAR acquisition tool is killed if the screen of a mobile device is off for a period of time, that causes the database application cannot receive data from the tool. The screen needs to be on during the collecting process which is inefficient. By studying the life cycle of the processes and application in Android, the database application currently does not have this problem, because it is a foreground process. As presented in the API guides for the life cycle of processes and application (Android, retrieved 2017[a]), Android puts each process into an "importance hierarchy" based on the components of the process and the state of these components; the first priority in the list are foreground processes which are only killed as a last resort when memory is too low. Therefore, nothing can guarantee that an application is never killed by the system.

The SQLite database management system, which is used for implementing the designed database model, is integrated in the Android platform, and does not allow multiple threads read the database in parallel. As presented in SQLiteOpenHelper API(Android, retrieved 2017[c]), the same object, which is a write database object, is returned regardless the get readable or get writable functions are called. Hence, using multiple reading threads to increase the database performance is not possible on the Android platform. Moreover, the SQLite uses dynamic types for storing data. That is, the data types need to be converted when saving and retrieving, and it is therefore causing poor performance.

Considerations concerning the design and implementation
The designs for the database model and database application are divided into two

parts, the first part is a high level and platform independent, while the second part is a specific implementation, and is considered a proof of concept for the high level design. Therefore, the application might contains many bugs, and not handle all small problems which can cause the application crashed. The implementation is mainly focusing on proving the feasibility of the database model and database application designs, and is far away to be a commercial application.

7.5 Future works

This thesis provides a relational database model and a database application for storing OSA signals, and opens for storing other bio-physiological signals on a mobile device. In addition, the designs are platform independent, and they can therefore be implemented on either Android, iOS, Windows Phone, etc.

A suggestion for further research is to implement the relational database design on different mobile operations, then make a comparison between the implementations with respect to the experiments that are presented in the evaluation chapter. In addition, the database application provides a raw query interface that allows researchers to execute their SQL query statements. The researchers can also write their own mining modules as independent Android activities, then attach the modules to the last fragment in the database application. Hence, an other suggestion is to develop and evaluate different mining methods on the designed data for detecting if a patient has OSA problem. Nonetheless, a quality comparing between sensors can be performed by using multiple sensor sources to collect data simultaneously.

Beside the presented suggestions above, there are also some issues that need to be further investigated based on the current implementation of the database application. The first issue is the need to backup collected data to a external storage place such as a cloud server when the free space on the mobile device is at a specific threshold. Hence, it is necessary to have a thread which takes responsibility for backing up the collected data by exporting the old data to EDF files and send to a storage place. This thread does not need to keep running all the time, it is created when the size of the database reaches the threshold. The second issue is the need for remotely querying. That is, physicians are allowed remotely querying data on the mobile device of a patient, or can get some part of data by sending a request to the database application which is installed on patient's mobile device.

Appendix A

User manual for the database application

The source code for the application can be downloaded at **https://github.com/vietttifi/MasterThesisUiO** and import to Android Studio in order to install the application to a Android device. The application has four main fragments as presented in Figure A.1, in which Sub figureA.1(a) is used to manage real time sources, Sub figureA.1(b) for importing EDF/EDF plus files, Sub figure A.1(c) for real time visualization, and Sub figure A.1(d) for database tasks and replay data from the database.

A.1 Real time wrapper

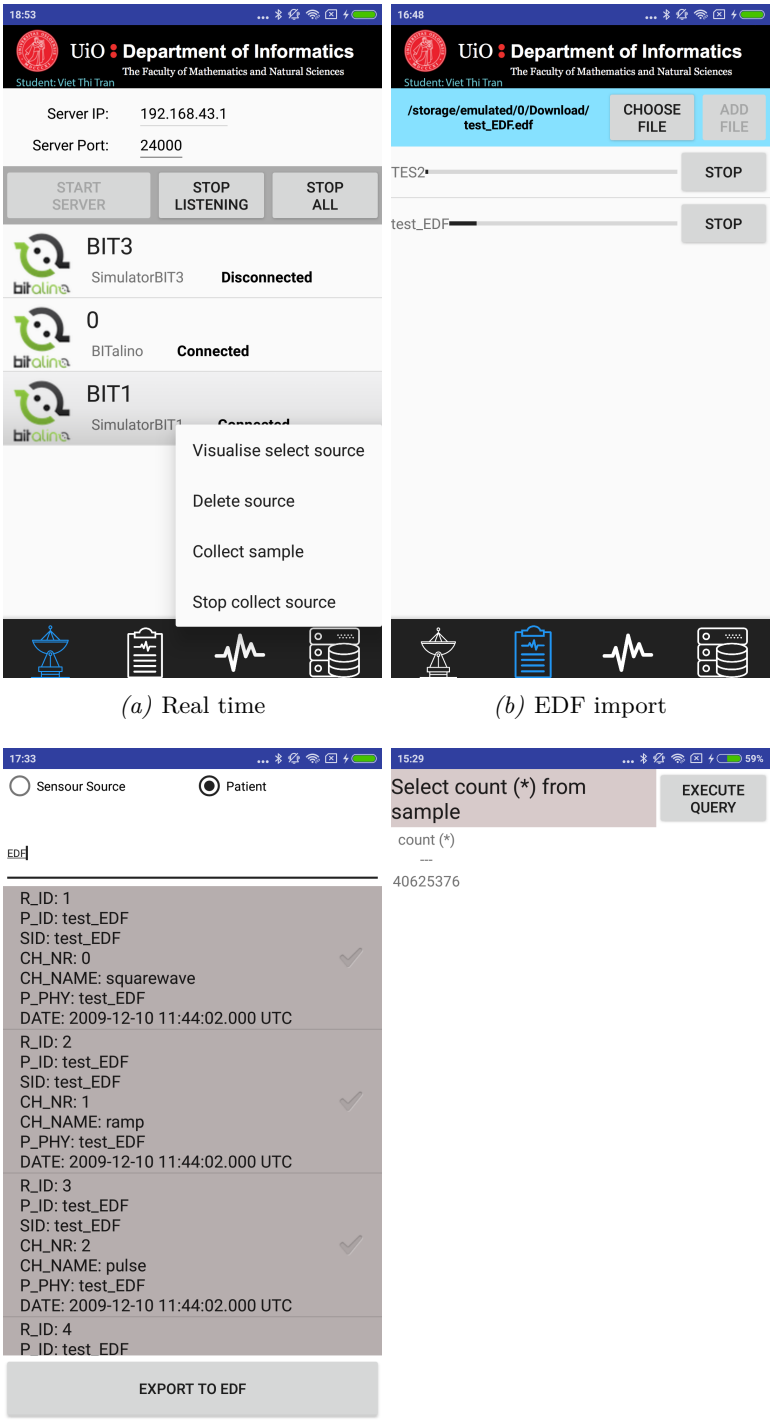
As presented in Sub figure A.1(a), the IP address and port number in which the device listens to are shown. Sensor sources can send data to the device by using the provided address. The port number is freely chosen, but not smaller than 1024. By long click on a source in the list, fours provided functions for the source are shown, which are visualization, source deletion, stop the collection, and begin a collection. Each source has a status which is either ”connected”, ”storing”, ”plotting”, ”plotting and storing” or ”disconnected” that presents a current status of the source.

A.2 Import EDF files

A file chooser is presented after a user clicks on ”choose file” button that lets the user choose a EDF/EDF plus file to read. By repeating ”choose file” and ”add file” actions, the user can simultaneously add multiple file. The user can stop the reading process by clicking on ”stop” button.

A.3 Visualize samples

For real time sources, a user must choose a source from the source list of the real time fragment to visualize. The application currently supports mono-source visualization, but users can choose which channels in the source that they want to visualize.



(a) Real time *(b)* EDF import

(c) EDF export *(d)* Database tasks

FIGURE A.1: Main fragments of the database application

For non real time sources, a user must query a record based on either patient, or sensor source information. By clicking on ”apply” button, the user can play or pause the visualization. When pausing the visualization, the user can add and save annotations to the database. However, to save annotation is currently not supported in this thesis, because data analyses are not the main focus of the thesis, but it is not so difficult to implement.

A.4 Export EDF files

Since data records are stored separately for each channel, by choosing data records that have the same patient, physician, and timestamp information, a full record for a patient at a specific time is retrieved. Users does not need to select all data records of the full record to export to a EDF/EDF plus file, they can select only the data records for specific interested channels to export.

A.5 Raw query

Users can execute their customized SQL queries by summiting the queries to the text box that presented in Sub figure A.1(d). Since the application does not filter out the vulnerable queries such as DELETE or DROP TABLE, users must be responsible for the queries they submit to the application.

Appendix B

Results of the experiments

This appendix presents the results from the experiments in Evaluation chapter. For each experiment, a graph of percentage of CPU usage is presented. Figures of the percentage of power are also presented if any. The other results are found on the github.com by following this link https://github.com/vietttifi/MasterThesisUiO/tree/master/MSc%20Latex/Experiments%20Results.

B.1 Real-time data collection experiment

B.2 Overnight experiment

B.3 EDF import

B.4 EDF export

B.5 Querying data experiment

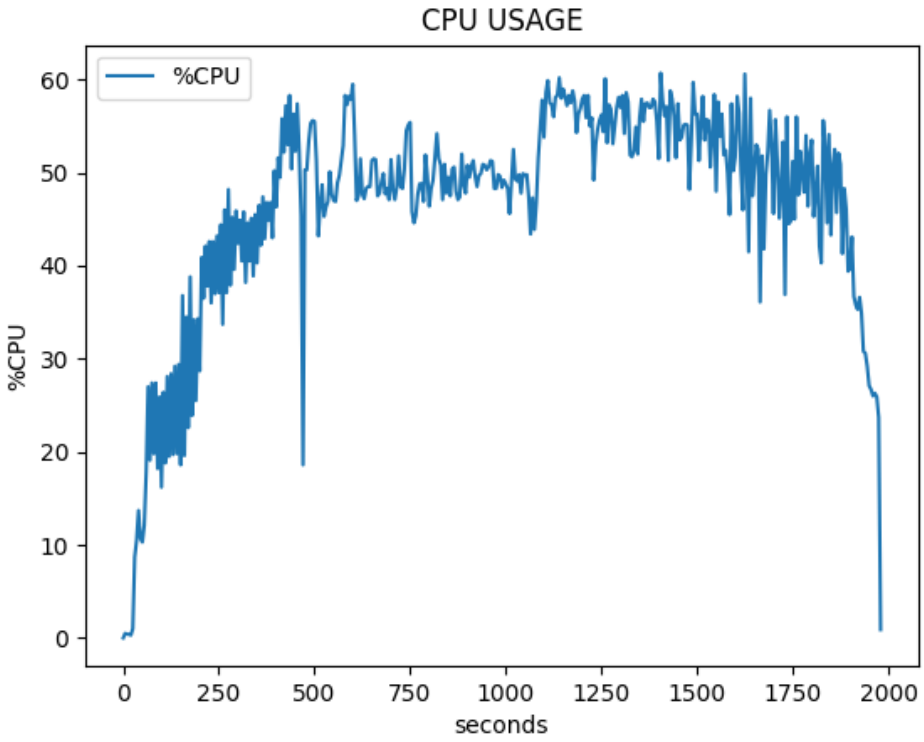


FIGURE B.1: CPU usage under stress test

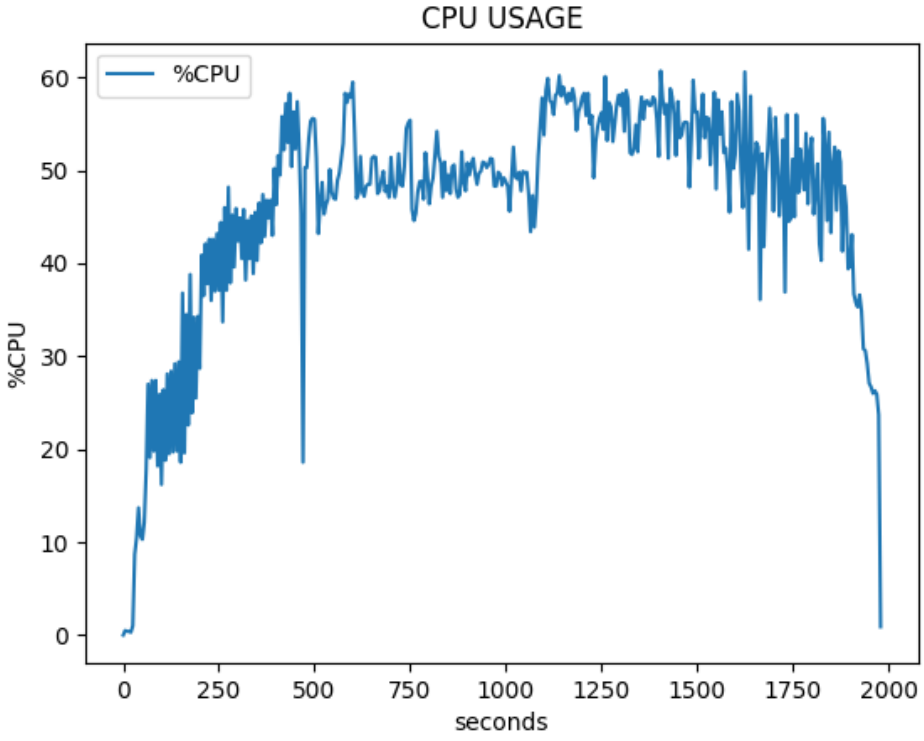


FIGURE B.2: CPU usage under stress test

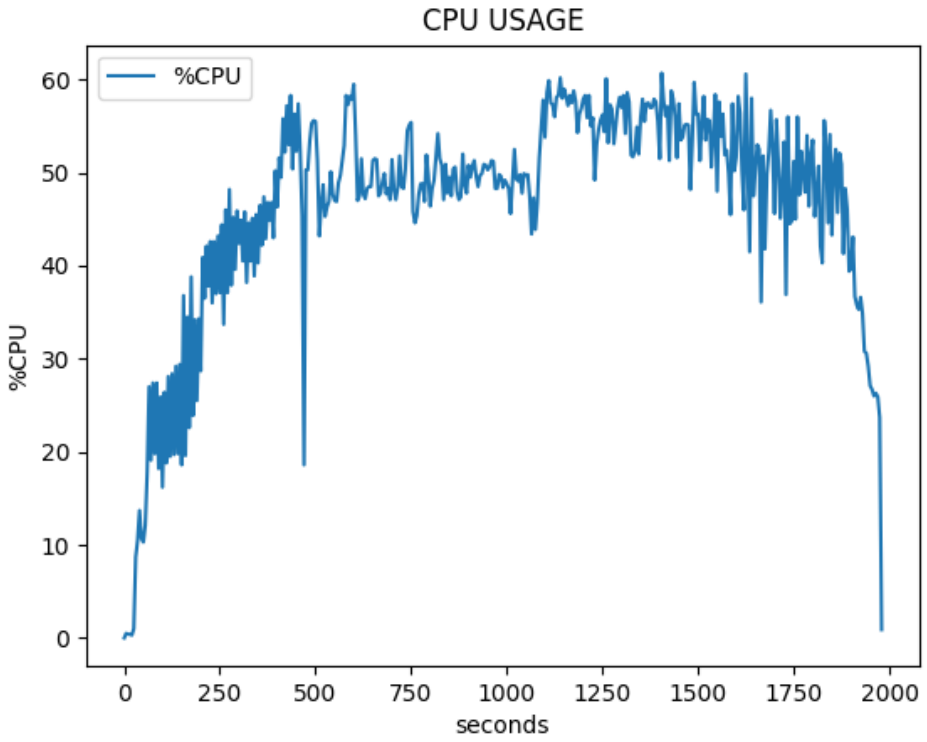


FIGURE B.3: CPU usage under stress test

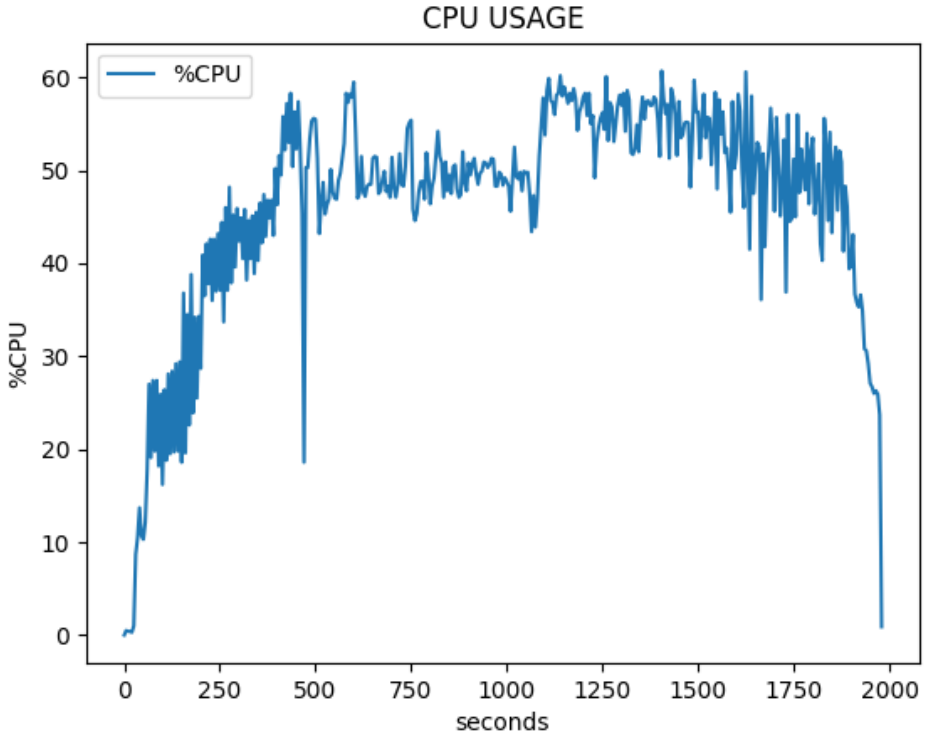


FIGURE B.4: CPU usage under stress test

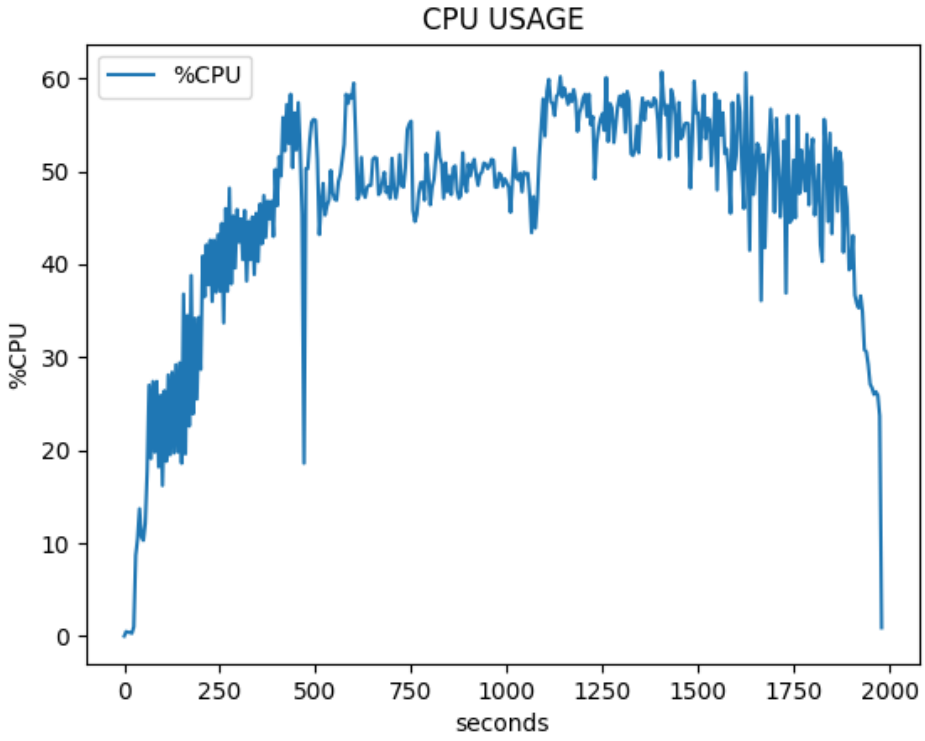


FIGURE B.5: CPU usage under stress test

Appendix C

Python code for parsing CPU usage

C.1 How to parse the output from Busybox

```
1  [H[JMem: 2720896K used, 39068K free, 0K shrd, 2496K buff, 537K cached
2  CPU: 27.2% usr  3.6% sys  0.0% nic 67.7% idle  0.2% io  0.6% irq  0.3%
       sirq
3  Load average: 6.89 6.25 5.88 3/2638 12713
4  [7m  PID  PPID USER     STAT   VSZ %VSZ CPU %CPU COMMAND[0m
5  11161   769 app_199  S <  2310m 85.3   0 25.6 {i.viettt.mscosa} no.uio.
       ifi.viett
6    585     1 system   R <   133m  4.9   1  1.8 /system/bin/
         surfaceflinger
7   1521   769 system   S    2530m 93.4   0  1.1 system_server
8  10096     2 root     SW       0  0.0   0  0.3 [kworker/u8:3]
9    602     1 system   S    68152  2.4   2  0.3 /system/vendor/bin/mm-pp-
         dpps
10 12592     2 root     DW       0  0.0   0  0.3 [mdss_fb0]
11   578     1 system   S     9424  0.3   0  0.3 /system/bin/
         servicemanager
12 10861     2 root     DW       0  0.0   0  0.3 [kworker/u8:5]
13     7     2 root     SW       0  0.0   3  0.2 [rcu_preempt]
14  2021   769 system   S <  2417m 89.2   2  0.1 {ndroid.systemui} com.
       android.syst
15 12706 23916 shell    R     1416  0.0   3  0.1 busybox top d 5
16 .....................
```

Listing C.1: A sample of five seconds fragment output from the text file

To get the percent CPU usage from the text files, a pattern for identifying the percent numbers must be identified. As presented in Listing C.1, the numbers are always in between a space and an other space following by {i.viettt.mscosa}. Hence, all percent CPU usage for each 5 second fragment can be gotten by scanning the text files based on the pattern.

C.2 Python code for parsing

Python is a high-level programming language that is a good candidate for parsing text and drawing charts. To parse and draw charts from the text files, three libraries are used, which are re (regular expression), numpy (the fundamental package for scientific computing with Python), and matplotlib.pyplot (drawing chart). The text files are read into buffers, then the pattern is applied on each buffer. The results from applying the pattern are presented on a chart by using pyplot library. The python code for parsing and drawing the percent CPU usages is presented in C.2.

```
1  import sys
2  import re
3  import numpy
4  import matplotlib.pyplot as pp
5
6  if __name__ == "__main__":
7      #Get param from terminal
8      filetext = sys.argv[1]
9      filetext1 = sys.argv[2]
10     filetext2 = sys.argv[3]
11
12     f = open(filetext)
13     content_source = f.read()
14     f1 = open(filetext1)
15     content_source1 = f1.read()
16     f2 = open(filetext2)
17     content_source2 = f2.read()
18     f.close()
19     f1.close()
20     f2.close()
21
22     matches = [float(i) for i in re.findall('\S[\d+\.]*\d+(?=\s+\{i\.
           viettt)', content_source, re.DOTALL)]
23     matches1 = [float(i) for i in re.findall('\S[\d+\.]*\d+(?=\s+\{i\.
           viettt)', content_source1, re.DOTALL)]
24     matches2 = [float(i) for i in re.findall('\S[\d+\.]*\d+(?=\s+\{i\.
           viettt)', content_source2, re.DOTALL)]
25
26     print(numpy.mean(matches))
27     print(numpy.mean(matches1))
28     print(numpy.mean(matches2))
29
30     pp.plot(numpy.arange(0, len(matches)*5,5), matches)
31     pp.plot(numpy.arange(0, len(matches1)*5,5), matches1)
32     pp.plot(numpy.arange(0, len(matches2)*5,5), matches2)
33
34     pp.legend(['buffer␣10s', 'buffer␣20s', 'buffer␣30s'], loc='upper␣
           left')
35     pp.xlabel('seconds')
36     pp.ylabel('%CPU')
37     pp.title('CPU␣USAGE')
38     pp.show()
```

Listing C.2: Python code for parsing CPU usage

Appendix D

SQLite code for creating tables

Listing D.1: SQLite code for creating table SensorSource

```
1  CREATE TABLE SENSORSOURCE(
2          SOURCE_ID               TEXT PRIMARY KEY,
3          SOURCE_NAME             TEXT,
4          SOURCE_TYPE             TEXT
5      );
```

Listing D.2: SQLite code for creating table Patient

```
1  CREATE TABLE PERSON(
2          PERSON_ID               TEXT PRIMARY KEY,
3      PERSON_NAME             TEXT,
4      PERSON_CITY             TEXT,
5      PERSON_PHONE            TEXT,
6      PERSON_EMAIL            TEXT,
7      PERSON_GENDER           TEXT,
8      PERSON_DAY_OF_BIRTH     TEXT,
9      PERSON_AGE              INTEGER
10 );
```

Listing D.3: SQLite code for creating table Patient

```
1  CREATE TABLE PATIENT(
2      PATIENT_PER_ID          TEXT NOT NULL,
3      PATIENT_CLINIC_P        TEXT NOT NULL,
4      PATIENT_PATIENT_NR      TEXT,
5      PATIENT_HEIGHT          TEXT,
6      PATIENT_WEIGHT          TEXT,
7      PATIENT_BMI             TEXT,
8      PATIENT_HEALTH_ISSUES   TEXT,
9      PRIMARY KEY (PATIENT_PER_ID , PATIENT_CLINIC_P),
10     UNIQUE (PATIENT_CLINIC_P ,PATIENT_PATIENT_NR),
11     FOREIGN KEY(PATIENT_PER_ID) REFERENCES
          TABLE_PERSON (PERSON_ID) ON DELETE CASCADE,
```

```
12          FOREIGN KEY(PATIENT_CLINIC_P) REFERENCES
                TABLE_CLINIC (CLINIC_ID) ON DELETE CASCADE
13      );
```

Listing D.4: SQLite code for creating table Physician

```
1       CREATE TABLE PHYSICIAN(
2           PHY_PERSON_ID           TEXT NOT NULL,
3           PHY_CLINIC_ID           TEXT NOT NULL,
4           PHY_EMPLOYEE_NR         TEXT,
5           PHY_TITLE               TEXT,
6           PRIMARY KEY (PHY_PERSON_ID, PHY_CLINIC_ID),
7           UNIQUE (PHY_CLINIC_ID,PHY_EMPLOYEE_NR),
8           FOREIGN KEY(PHY_PERSON_ID) REFERENCES TABLE_PERSON
                (PERSON_ID) ON DELETE CASCADE,
9           FOREIGN KEY(PHY_CLINIC_ID) REFERENCES TABLE_CLINIC
                (CLINIC_ID) ON DELETE CASCADE
10      );
```

Listing D.5: SQLite code for creating table Clinic

```
1       CREATE TABLE CHANNEL(
2           CLINIC_ID          TEXT PRIMARY KEY,
3           CLINIC_NAME        TEXT,
4           CLINIC_ADDRESS     TEXT,
5           CLINIC_PHONE_NR    TEXT,
6           CLINIC_EMAIL       TEXT
7       );
```

Listing D.6: SQLite code for creating table Record

```
1       CREATE TABLE RECORD(
2           RECORD_ID              INTEGER PRIMARY KEY
                AUTOINCREMENT,
3           RECORD_S_ID            TEXT NOT NULL,
4           RECORD_CH_NR           INTEGER NOT NULL,
5           RECORD_PHYSICIAN_ID    TEXT NOT NULL,
6           RECORD_PATIENT_ID      TEXT NOT NULL,
7           RECORD_TIMESTAMP       INTEGER NOT NULL,
8           RECORD_DESCRIPTIONS    TEXT,
9           RECORD_FREQUENCY       REAL,
10          RECORD_USED_EQUIPMENT  TEXT,
11          RECORD_EDF_RESERVED    BLOB,
12          UNIQUE (RECORD_S_ID,RECORD_CH_NR, RECORD_TIMESTAMP
                , RECORD_PHYSICIAN_ID, RECORD_PATIENT_ID),
```

```
13          FOREIGN KEY(RECORD_S_ID, RECORD_CH_NR) REFERENCES
               TABLE_CHANNEL (CHANNEL_S_ID, CHANNEL_NR) ON
               DELETE CASCADE,
14          FOREIGN KEY(RECORD_PHYSICIAN_ID) REFERENCES
               TABLE_PHYSICIAN (PHY_PERSON_ID) ON DELETE
               CASCADE,
15          FOREIGN KEY(RECORD_PATIENT_ID) REFERENCES
               TABLE_PATIENT (PATIENT_PER_ID) ON DELETE
               CASCADE
16      );
```

Listing D.7: SQLite code for creating table RecordAnnotation

```
1       CREATE TABLE RECORDANNOTATION(
2           RECORD_ANNOTATION_ID     INTEGER NOT NULL,
3           RECORD_ANNOTATION_R_ID   INTEGER NOT NULL,
4           PRIMARY KEY (RECORD_ANNOTATION_ID,
               RECORD_ANNOTATION_R_ID),
5           FOREIGN KEY (RECORD_ANNOTATION_R_ID) REFERENCES
               TABLE_RECORD (RECORD_ID) ON DELETE CASCADE,
6           FOREIGN KEY (RECORD_ANNOTATION_ID) REFERENCES
               TABLE_ANNOTATION (ANNOTATION_ID) ON DELETE
               CASCADE
7       );
```

Listing D.8: SQLite code for creating table Annotation

```
1       CREATE TABLE ANNOTATION(
2           ANNOTATION_ID           INTEGER PRIMARY KEY
               AUTOINCREMENT,
3           ANNOTATION_ONSET        INTEGER NOT NULL,
4           ANNOTATION_DURATION     REAL,
5           ANNOTATION_TIMEKEEPING  INTEGER,
6           ANNOTATION_TEXT         TEXT
7       );";
```

Listing D.9: SQLite code for creating table Sample

```
1       CREATE TABLE SAMPLE(
2           SAMPLE_RECORD_ID      INTEGER NOT NULL,
3           SAMPLE_TIMESTAMP      INTEGER NOT NULL,
4           SAMPLE_VALUE          REAL NOT NULL,
5           PRIMARY KEY (SAMPLE_RECORD_ID, SAMPLE_TIMESTAMP),
6           FOREIGN KEY (SAMPLE_RECORD_ID) REFERENCES
               TABLE_RECORD (RECORD_ID) ON DELETE CASCADE
```

```
7       );
```

Bibliography

American_Academy_of_Sleep_Medicine (2005). "The international classification of sleep disorders : diagnostic and coding manual". In: *Obstructive Sleep Apnea Syndrome*, p. 52.

Android (2017a). "android.database.sqlite". In: URL: https://developer.android.com/reference/android/database/sqlite/package-summary.html.

— (2017b). "Dashboards". In: URL: https://developer.android.com/about/dashboards/index.html.

— (retrieved 2017[a]). "Processes and Application Life Cycle". In: URL: https://developer.android.com/guide/topics/processes/process-lifecycle.html.

— (retrieved 2017[b]). "Processes and Threads". In: URL: https://developer.android.com/guide/components/processes-and-threads.html.

— (retrieved 2017[c]). "SQLiteOpenHelper". In: URL: https://developer.android.com/reference/android/database/sqlite/SQLiteOpenHelper.html.

Babak Mokhlesi, et.al (2012). ""REM-related" Obstructive Sleep Apnea: An Epiphenomenon or a Clinically Important Entity?" In: URL: http://www.journalsleep.org/ViewAbstract.aspx?pid=28396.

Bersain Reyes, et.al (February 25, 2016). "Tidal Volume and Instantaneous Respiration Rate Estimation using a Smartphone Camera". In: URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=7415992.

BITalino (2016a). "BITalino (r)evolution Board Kit Data Sheet". In: URL: http://bitalino.com/datasheets/REVOLUTION_BITalino_Board_Kit_Datasheet.pdf.

— (2016b). "BITalino (r)evolution Freestyle Kit Data Sheet". In: URL: http://bitalino.com/datasheets/REVOLUTION_BITalino_Freestyle_Kit_Datasheet.pdf.

— (2016c). "BITalino (r)evolution Plugged Kit Data Sheet". In: URL: http://bitalino.com/datasheets/REVOLUTION_BITalino_Plugged_Kit_Datasheet.pdf.

— (2016d). "Disclaimer". In: URL: http://plux.info/index.php/en/policies/119-privacy-policy.

— (2016e). "Microcontroller Unit (MCU) Block Data Sheet". In: URL: http://bitalino.com/datasheets/REVOLUTION_MCU_Block_Datasheet.pdf.

— (2016f). "Publications". In: URL: http://www.bitalino.com/index.php/community/publications.

Carlos Carreiras, et.al (2011). "Logical Sensor Specification". In: URL: http://www.cs.utah.edu/~tch/publications/pub30.pdf.

C.Hansen, et.al (1984). "Logical sensor specification". In: pp. 169–193. URL: http://www.cs.utah.edu/~tch/publications/pub38.pdf.

Charles Hansen, et.al (2011). "StorageBIT A Metadata-aware, Extensible, Semantic, and Hierarchical Database for Biosignals". In: URL: http://www.lx.it.pt/~afred/papers/StorageBIT.pdf.

Daniel de Sousa Michels, et.al (2014). "Nasal Involvement in Obstructive Sleep Apnea Syndrome". In: URL: http://www.hindawi.com/journals/ijoto/2014/717419/.

edfplus.info. "EDF INFO". In: URL: http://www.edfplus.info.

— "Full specification of EDF". In: URL: http://www.edfplus.info/specs/edf.html.

— "Full specification of EDF+". In: URL: http://www.edfplus.info/specs/edfplus.html.

Emanuele_Panigati, et.al (2017). "Data Streams and Data Stream Management Systems and Languages". In: URL: http://eecs.wsu.edu/~yinghui/mat/courses/spring%202016/Reading/chp5-data%20stream%20management.pdf.

Evgenij_Thorstensen (2017). "Database system course". In: URL: http://www.uio.no/studier/emner/matnat/ifi/INF3100/v17/.

George B. Moody, et.al. "PhysioNet: A Web-Based Resource for the Study of Physiologic Signals". In: URL: http://ecg.mit.edu/george/publications/physionet-embs-2001.pdf.

George, B. Moody. "WFDB Programmer's Guide". In: URL: https://www.physionet.org/physiotools/wpg/.

Gjørby, Svein Petter (2016). "Generic data acquisition for mobile platforms, the CESAR acquisition toll". In:

Graeme, et.al (retrieved 2017). "A SQL Database for Sensor Nodes and Embedded Applications". In: URL: http://ai2-s2-pdfs.s3.amazonaws.com/e1d6/7eac67173b03966b4d038bdfaa1f6acd0986.pdf.

Grapview (2017). "Graph View Android". In: URL: http://www.android-graphview.org.

Grapview_code (2017). "LineGraphSeries java file". In: URL: https://github.com/appsthatmatter/GraphView/blob/master/src/main/java/com/jjoe64/graphview/series/LineGraphSeries.java.

Halpin, Terry (2017a). "Object-Role Modeling". In: URL: http://www.orm.net.

— (2017b). "ORM 2". In: URL: http://www.orm.net/pdf/ORM2.pdf.

Harald Hrubos-Strøm, et.al (16 June 2010). "A Norwegian population-based study on the risk and prevalence of obstructive sleep apnea The Akershus Sleep Apnea Project (ASAP)". In: URL: http://onlinelibrary.wiley.com/doi/10.1111/j.1365-2869.2010.00861.x/full.

heart.org (April, 2016). "All About Heart Rate (Pulse)". In: URL: http://www.heart.org/HEARTORG/Conditions/More/MyHeartandStrokeNews/All-About-Heart-Rate-Pulse_UCM_438850_Article.jsp#.V9boyTt1ZVo.

— (August, 2016). "Understanding Blood Pressure Readings". In: URL: http://www.heart.org/HEARTORG/Conditions/HighBloodPressure/AboutHighBloodPressure/Understanding-Blood-Pressure-Readings_UCM_301764_Article.jsp#.V7sFAWWDBug.

— (June 2001). "Sleep as a teaching tool for integrating respiratory physiology and motor control". In: URL: http://advan.physiology.org/content/25/2/29.

Helsenorge.no (2014). "Snorking og søvnapné". In: URL: https://helsenorge.no/sykdom/sovnproblemer/snorking-og-sovnapne.

Ian, Sommerville (2017). "Software Engineering". In:

IANA (2017). "service names and port numbers". In: URL: https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.txt.

Karakis, et.al (August 29, 2012). "The utility of routine EEG in the diagnosis of sleep disordered breathing." In: URL: http://www.ncbi.nlm.nih.gov/pubmed/22854767.

Laiali Almazaydeh, et.al (May 2012). "Detection of Obstructive Sleep Apnea Through ECG Signal Features". In: URL: https://www.researchgate.net/publication/254039441_Detection_of_obstructive_sleep_apnea_through_ECG_signal_features.

Marc B Blumen, et.al (May 20, 2004). "Tongue mechanical characteristics and genioglossus muscle EMG in obstructive sleep apnoea patients". In: URL: http://www.sciencedirect.com/science/article/pii/S1569904803002933.

mayoclinic.org (June 15, 2016). "Obstructive sleep apnea - Treatment". In: URL: http://www.mayoclinic.org/diseases-conditions/obstructive-sleep-apnea/diagnosis-treatment/treatment/txc-20206034.

Mihaela, et.al (04.2016). "Smoking, Drinking, Overweight, and Obstructive Sleep Apnea Among Patients With Sleep Breathing Disorders". In: URL: http://journal.publications.chestnet.org/data/Journals/CHEST/935163/02594.pdf.

MIOB (2017). "Java-Parser for the file formats EDF and EDF+". In: URL: https://github.com/MIOB/EDF4J.

Mons Universite, de (retrieved 2017). "The DREAMS Apnea Database". In: URL: http://www.tcts.fpms.ac.be/~devuyst/Databases/DatabaseApnea/.

Muhammad, et.al (February 16, 2014). "Complex Sleep Apnea Syndrome". In: URL: http://www.hindawi.com/journals/sd/2014/798487/.

Munthe-Kaas, Ellen (2017). "Recipe book". In: URL: http://www.uio.no/studier/emner/matnat/ifi/INF3100/v16/undervisningsmateriale/forelesningsmateriale/oppskriftsbok.pdf.

Naresh, et.al (2008). "The Epidemiology of Adult Obstructive Sleep Apnea". In: URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2645248/pdf/PROCATS52136.pdf.

netmarketshare.com (2017). "operating system market share". In: URL: https://www.netmarketshare.com/operating-system-market-share.aspx?qprid=8&qpcustomd=1.

nhlbi (May 2009). "Sleep Apnea: What is sleep apnea?" In: URL: http://www.nhlbi.nih.gov/health/health-topics/topics/sleepapnea/.

Nicolas, et.al (retrieved 2017). "A Database in Every Sensor". In: URL: http://dunkels.com/adam/tsiftes11database.pdf.

Payam Barnaghi, et.al. "Computing Perception from Sensor Data". In: URL: http://personal.ee.surrey.ac.uk/Personal/P.Barnaghi/doc/PID2479545.pdf.

PhysioNet. "Contributing to PhysioNet - Data Contributions". In: URL: https://physionet.org/guidelines.shtml.

— "Training Opportunities at PhysioNet". In: URL: https://www.physionet.org/training.shtml.

Physionet. "Apnea-ECG Database Annotations". In: URL: https://physionet.org/physiobank/database/apnea-ecg/annotations.shtml.

— "Data for development and evaluation of ECG-based apnea detectors". In: *CinC Challenge 2000 data sets*. URL: https://physionet.org/physiobank/database/apnea-ecg/.

— "Subjects". In: *St. Vincent's University Hospital / University College Dublin Sleep Apnea Database*. URL: https://physionet.org/pn3/ucddb/.

— "Subjects". In: *St. Vincent's University Hospital / University College Dublin Sleep Apnea Database*. URL: https://physionet.org/pn3/ucddb/#annotations.

— "The Apnea-ECG Database". In: *Computers in Cardiology 2000*. URL: https://physionet.org/challenge/2000/.

physionet.org. "PhysioBank ATM". In: URL: https://www.physionet.org/cgi-bin/atm/ATM?tool=&database=shhpsgdb&rbase=&srecord=&annotator=&signal=&sfreq=&tstart=&tdur=&tfinal=&action=&tfmt=&dfmt=&nbwidth=.

sciencedaily.com (September 4, 2006). "Mayo Clinic Discovers New Type Of Sleep Apnea". In: URL: https://www.sciencedaily.com/releases/2006/09/060901161349.htm.

Sleepdata.org. "Sleep Heart Health Study". In: URL: https://sleepdata.org/datasets/shhs/.

sleepdata.org. "National Sleep Research Resource". In: URL: https://sleepdata.org.

Sleep_Medicine (2009). "Prevalence of undiagnosed obstructive sleep apnea among adult surgical patients in an academic medical center". In: URL: http://www.sciencedirect.com/science/article/pii/S1389945708003079.

sleeptmjdr.com (2016). "Obstructive sleep apnea". In: URL: http://sleeptmjdr.com/obstructive-sleep-apnea/.

SQLite. "About SQLite". In: URL: https://www.sqlite.org/about.html.

— (2017a). "Cross join". In: URL: https://sqlite.org/lang_select.html#crossjoin.

— (2017b). “Insert performance”. In: URL: http://www.sqlite.org/faq.html#q19.

teuniz.net (2017). “EDFbrowser and EDFLib”. In: URL: http://www.teuniz.net/edfbrowser/.

Thalheim, David W. Embley Bernhard (2011). “Handbook of Conceptual Modeling”. In:

Thomas, Line (August 26th 2016). “Email communication between Line Michelsen (Resmed) and Thomas Plagemann (University of Oslo)”. In:

Toby Teorey Sam Lightstone, Tom Nadeau (2006). “Database Modeling & Design: Logical Design”. In:

Tom Henderson, et.al (1984). “Logical Sensor System”. In: URL: http://www.cs.utah.edu/~tch/publications/pub38.pdf.

T_Penzel, et.al (2017). “Data for development and evaluation”. In: URL: http://ecg.mit.edu/george/publications/apnea-ecg-cinc-2000.pdf.

veracode.com (2017). “Owasp top 10 vulnerabilities”. In: URL: https://www.veracode.com/directory/owasp-top-10.

Wafaa S.Almuhammadi, et.al (May 1, 2015). “Efficient Obstructive Sleep Apnea Classification Based on EEG Signals”. In: URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7160186.

wikihow.com (August, 2016). “How to Measure Oxygen Saturation Using Pulse Oximeter”. In: URL: http://www.wikihow.com/Measure-Oxygen-Saturation-Using-Pulse-Oximeter.

Wikipedia (2016). “Obstructive sleep apnea”. In: URL: https://en.wikipedia.org/wiki/Obstructive_sleep_apnea.

— (2017a). “Object-role modeling”. In: URL: https://en.wikipedia.org/wiki/Object-role_modeling.

— (2017b). “Producer and consumer problem”. In: URL: https://en.wikipedia.org/wiki/Producer%E2%80%93consumer_problem.

Y.Ichimaru, et.al. “Development of the polysomnographic database on CD-ROM”. In: URL: http://ecg.mit.edu/george/publications/slpdb-pcn-1999.pdf.