FACULTY OF SCIENCE AND TECHNOLOGY

Department of Computer Science

Data Management for Nudged Green Transportation

—

Cosmin Radu Crăciun

INF-3990 Master's Thesis in Computer Science

May 2019

Declaration

I, Cosmin Radu Crăciun, hereby declare that this thesis in its entirety has been composed by myself and has not been submitted, in whole or part for any previous degree or professional qualification. However, I have been part of the Open Distributed Systems (ODS) research group, working in close collaboration with Anders Andersen and Randi Karlsen as supervisors and Jemea Lady Limunga who worked on the Data Analysis part of this implementation. Any other form of information or inspiration gotten from other people's work has been well referenced.

Abstract

Climate change is one of the most talked about topics in the world at the moment. In the context of man induced Global Warming, there are many proposed ideas on how to combat its effects and many more are still needed. We propose employing nudge theory to persuade people into using environmentally friendly modes of transport through a software application. This thesis focuses on the data management part of such an application, mainly on storing and providing data from persistent storage or other sources. My approach involves using approximate query processing and a key-value in-memory data store to optimise data access.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ACID – atomicity, consistency, isolation, durability

API – Application Programming Interface

APQ – Approximate Query Processing

BASE – Basically Available, Soft state, Eventually consistent

CAP – consistency, availability, partition tolerance

CO_2 – Carbon Dioxide

CSS – Cascading Style Sheets

DRAM – Dynamic Random-Access Memory

EF – Environmental Friendliness

EF Core – Entity Framework Core

HTTP – Hypertext Transfer Protocol

IDE - Integrated Development Environment

JSON – JavaScript Object Notation

LINQ – Language Integrated Query

NoSQL – Not only SQL

OLAP – On-Line Analytical Processing

OLTP – On-Line Transaction Processing

PB – Petabytes

RDMA – Remote Direct Memory Access

RDMS – Relational Database Management System

REST – Representational state transfer

SPA – Single Page Application

SQL – Structured Query Language

SSD – Solid State Drive

XML – eXtensible Markup Language

1. Introduction

1.1. Motivation

Anthropological induced global warming is an important topic nowadays. The high emission of greenhouse gasses has a significant impact on the environment and on human health. We can already see the effects of it in recent years with more extreme weather events. It is predicted that these events will worsen, and new problems will arise.

One of the biggest culprits of global warming is considered the transportation sector, which includes all modes of transport: aerial, land or naval, both private and commercial. Moreover, people driving cars contribute to the problem themselves as cars are considered one of the least environmentally friendly mode of transport promoting congestions in cities, which reduces the air quality and increases frustration among commuters.

Based on these considerations, our motivation is to do our part to help reducing those emissions, for forming a better future for next generations and helping people lead a healthier life.

1.2. Challenges

The main challenge is persuading people into changing the default way that they travel, convincing them to use public transport or walking instead of taking their personal car. People react differently to different situations and the challenges change depending on each person's needs and background.

For building a system that accomplishes this, we need to store massive amounts of heterogeneous data, analyse it appropriately and, depending on the results, send messages to people to push them towards greener mobility choices.

Implementing such an application involves other technical challenges, especially looking at data management when involving Big Data. We need to consider what data we need, what needs to be stored for historical purposes and what not to store. A serious challenge is how to access the data, being it historical or coming from external providers, while maintaining rapid response times. Accessing stored data in the context of Big Data requires big computational efforts that usually result in slow response times, which slows down data analysis.

Requesting the same data resources from external providers over a short timespan can make this process tedious as the results will be the same. This slows down the entire process, requiring either the user or other services to wait longer for a response.

Nudging for green transportation involves receiving heterogeneous data from multiple sources, converting it into a more aggregated form, and either using or storing it for later analysis. Traffic, weather forecast, user data, history and other environmental information need to be considered and converted into a form that is usable for analysis.

1.3. Goals

The goal of the project is to build an application that uses nudge theory to help people make greener mobility choices. My specific goal is handling the data management part of the application, focusing on data access and storage. This involves storing nudge specific information and accessing data from multiple heterogeneous data sources.

I hypothesize that using a relational database with approximate query processing capabilities will offer an enormous improvement, up to 2 times faster response times, over traditional querying. Also, I suggest that using an in-memory data store for storing certain data will offer improvements in accessing data from external providers.

1.4. Problem statement

Climate change is a serious issue that affects everyone, solutions to mitigate this problem are needed as soon as possible. This problem cannot be solved with a single solution, as it has global impact. Nudging through an application in order to change people's methods of transport to lower their environmental impact does not come without challenges. The data needed for a successful, personalised tailored nudge is diverse and extensive.

A good way to manage data that is tailored for personalised nudging is needed. The data needs to be obtained from different sources, filtered, combined and stored if necessary for later access. Accessing massive amounts of data can be slow, methods for fast access are required.

1.5. Approach

We built an application that helps people choose greener modes of transport by offering them better options to choose from and helping them change their traditional, default way of traveling, to one with a lower environmental impact. To achieve this, we nudge users when they need to go somewhere or simply based on the weather.

Our application sends nudge notifications to the user based on his and other users' history and based on external environmental data.

My approach is to store user and nudge information for each user in a relational database. For faster access to the data, I chose a database that supports approximate queries, as I expect the nudge data amount to grow very fast. Making approximate queries should improve the response times for data analysis support, while still giving valuable responses.

For handling the external data, we filter out the information we do not need from external data providers and present it to the user after conversion into more understandable forms. For example, we take the cloud coverage percentage and present it to the user in terms which are more easily to understand (sunny, cloudy, partly cloudy, etc.).

To maintain a faster response time for certain data that has fast change rates, I store some responses in memory for a short amount of time and serve the requests from there. I do this when getting forecast information, storing it for one hour before removing the data from memory. I define this kind of data transient, temporary data.

The sources that we propose to receive data from are: user's location, map information, the user's events calendar, national events calendar, weather forecast, current weather conditions, traffic data, travel route information and bus schedules.

We propose the next use cases for exemplifying the possible uses of our application:

- A user wants to go to a certain location. He provides us his location and his destination address. We provide him with different choices of transportation, trip duration and current weather forecast information.
- A new user has an appointment. New users don't have any historical data. We send requests for weather and bus information for the time of the appointment at the external providers. We do an analysis based on the response, the user's location and destination, and other users' history, then, we store the nudge information and then nudge the user.
- An old user has an appointment. We send requests for weather and bus information for the time of the appointment at the external providers. We also provide historical information about similar nudges. We do an analysis based on the data and other users in similar conditions, store the nudge information and then nudge result and then nudge the user.

1.6. Results

I run some experiments involving approximate query processing and caching with the in-memory data store and found that they provide some benefits to the application, but not as high as expected.

Counting approximately distinct users in the nudge data provided improvements to up to around 20% in response times, which is less than the expected 50% improvement.

Using an in-memory data store to hold transient data received from data providers yielded shorter response times for future requests, with low trade-offs in worst case scenarios, as long as we can fairly guarantee that the data did not change in the time since the last request.

1.7. Contribution

Our main contribution is a web application which uses nudge notifications that promotes a more sustainable mode of transport. Our application can be used to get information about different modes of traveling between the user's location to his destination or receive nudges with suggested modes of transport based on events in the calendar. I collect data from multiple data sources, storing some in the process.

My contribution with this thesis is implementing the data management service for the application, which handles data storage and access, modalities of storing user and nudge data, how to access data from external providers and how to manage it. I use a relational database for storing user and nudge data and I use approximate query processing methods for filtering and querying huge chunks of nudge data for providing an overview of how other users travel. I also use an in-memory key-value data store for temporary storing transient data, optimizing response times for some requests.

I do some database experiments by filling the database with millions of entries to show how much approximate queries can help in shortening response times. I also show that using caching to avoid making redundant calls over the internet yields huge improvements with low trade-offs, as long as we are careful on when to use them, so that the data is still valuable for the user.

I also worked on the implementation of the frontend and other areas such as the core application and the task scheduler service.

The application was built in a team, but this thesis reflects my own work. I worked on the data management side of the application, the setup and access of the databases, the data store and accessing data from external providers. I also worked on the scheduler, the push notification service and the setting up both the client-side and the server-side of the application. We worked as a team at data aggregation and on the client-side code, except the services related to data accessing, which I implemented. The data analysis part of the application is out of this thesis' scope and should be accessed in [1]. In this thesis, when stating "we" I am referring to work done together, in the team, as opposed to "I", when I'm referring to work that I've done myself.

1.8. Limitations

Due to time constraints and chosen approach there are some issues that are out of the scope of this thesis.

Security and privacy are not handled in our approach. The security part on our application is minimal, employing simple authentication and authorization. Privacy is mentioned in some design choices, but it isn't the focus of the thesis.

Due to time constraints we are not able of running the application on a mobile device, which in turn limits our capabilities, as a user needs to keep the application open in the browser for our processes to run in the background effectively.

1.9. Outline

This thesis is organized as follows:

- Chapter 2 – Background – I go in more depth for my motivation, looking at climate change and green transportation. I also present what is the nudge theory and explain how it works, by offering some nudging examples, both traditional and digital nudging, and also looking at some existing applications. Afterwards, I go through what is big data and some data management practices that exist in the scientific literature. I look into approximate query processing and key-value stores, check some code patterns and some related applications.
- Chapter 3 – Architecture and Design – I describe my approach for data management, the general architecture of the application and the database design. I have an overall look over all the systems of the application.
- Chapter 4 – Implementation – This chapter describes exactly how the application was built, what services we created and how they work, and what tools and technologies were used.
- Chapter 5 – Experimentation – I present some experiments regarding the database with approximate queries and with the in-memory key-value store, how the application manages to optimise queries and requests.
- Chapter 6 – Discussion – This chapter contains the discussion and ideas based on the implementation and the experimentation.
- Chapter 7 – Future work – I argue what remains to be done and what could have been done better.
- Chapter 8 – Conclusion – The chapter dives into the conclusion of my work.

2. Background

2.1. Climate change

Global warming and climate change in general, are growing issues nowadays and they are increasingly debated topics causing public concern. The consensus among the vast majority of scientists studying different aspects of climate is that the rate of change in the climate has been unusual during the past century compared to the last millennium. It is highly probable that the main cause of these changes are attributed to human activity, to the emissions of greenhouse gases [2]. Figure 1 shows the surface temperature anomalies for 4 decades, since the 1970s until the 2000s. As seen there, the anomalies have a growing trend over the decades and they seem to be more extreme in the north polar regions, Europe, and Asia.

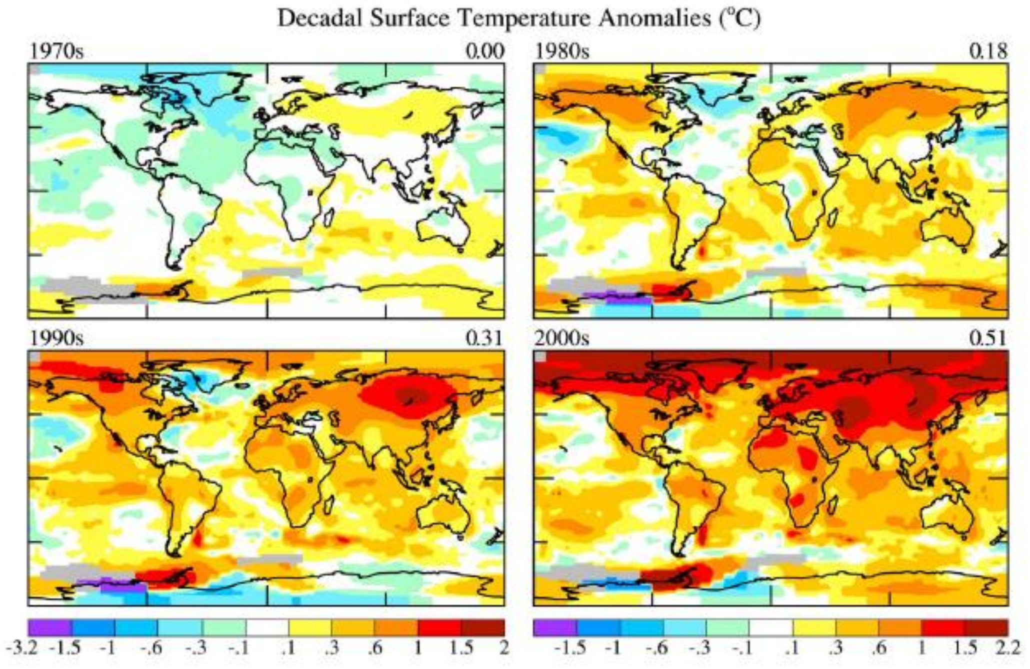


Figure 1 Decadal surface temperature anomalies relative to 1951–1980 base period. [3]

In the face of more extreme weather phenomena felt in multiple parts of the world, as seen as more severe and frequent droughts or storms, global warming seems a more serious threat than ever. Climate change increased the odds of a heat wave at least threefold since 1950. Warm surface temperatures in the Pacific Ocean doubled the probability of African drought, which contributes to food insecurities in the region. [4]

On the other hand, anthropogenic climate change has led to an increase in the frequency of intense tropical cyclones. Over the past 5000 years, in the western North Atlantic Ocean, the frequency of intense hurricane landfalls has increased tenfold. [5]

More extreme events can be seen in the polar regions, where the polar cap seems to be melting with increasing speed: in Northern Canada, the Queen Elizabeth Islands, the mean rate of glacier mass loss was 5 times greater between 2005 and 2009 than the 1963-2004 average, and even 7 times greater when considering only 2007 and 2008. [6]

The current predictions show that even if the concentration of greenhouse gases in the atmosphere would have stabilized in the year 2000, we would be already committed to global warming of another half degree global mean temperature and to rising sea levels, due to thermal expansion by the end of the 21st century [7]. This means that the coastal cities will be strongly affected by the rising of water levels, by dislocating a huge portion of the population living in them. Around 10% of the world population or 13% of the worlds' urban population is living in a coastal area – less than 10 meters above the sea level [8]. Due to the rising sea level trend, the dislocation of massive portions of the population will generate huge strains in the economy and in the society in general.

Technologies are being researched and big steps are made to solve this problem. The graph in figure 2 shows different technologies and it predicts how they would aid in reducing our CO_2 emissions by 2050, to limit global warming to an increase of 2^0 C.

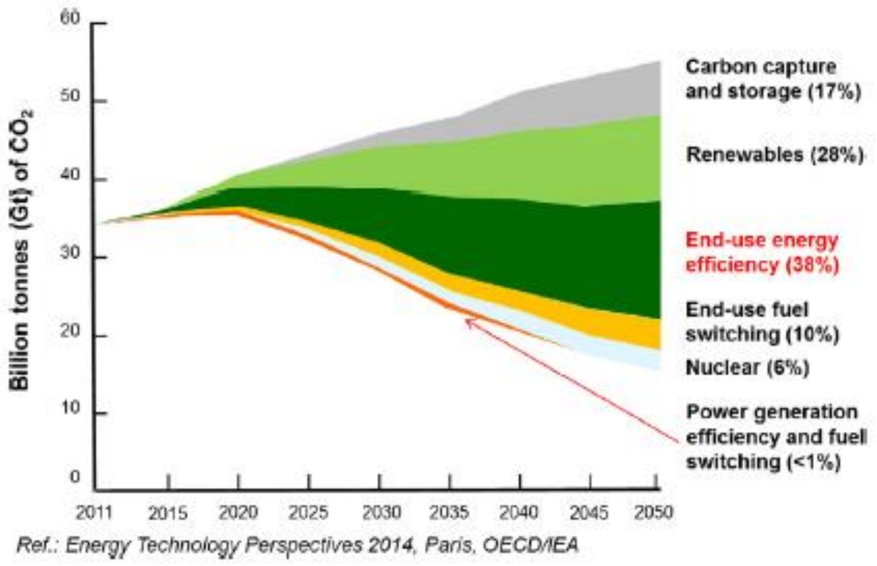


Figure 2 Key technologies for reduction of CO_2 emissions in order to limit global warming to 2^0 C [9]

Carbon Capture and Storage (CCS) is defined as the capture and secure storage of carbon that would otherwise be emitted or remain in the atmosphere [10]. This action makes it carbon negative, as the carbon is stored for thousands or millions of years. For example, Norway has been involved in CCS since before 2000. The project involves injecting liquefied carbon dioxide into a saline formation 2.6 km below the seabed. [11]

Producing biofuel is seen as environmentally friendly, as it is made from natural oils or fat combined with an alcohol. It can be made from oils such as rapeseed, or soybean. The fuel is used as a substitute for diesel fuel, called biodiesel [12]. This kind of technology comes closer of being carbon neutral, as it takes carbon from the atmosphere with the help of plants, and then using it to produce energy by burning the resulting fuel, putting the carbon back into the atmosphere.

The biggest culprit of today's accelerated climate change is considered to be the high emission of greenhouse gases generated by burning fossil fuel used for transportation, such as cars, trucks, trains or airplanes [13]. The pie chart in Figure 3 represents different industry sectors and their share in generating greenhouse gases in the atmosphere, according to the IEA (International Energy Agency) statistics for energy balance for 2004-2005. As seen here, the biggest chunk is occupied by the transportation sector (30.3%), followed by the industrial sector (29%) and then the residential sector (27.1%).

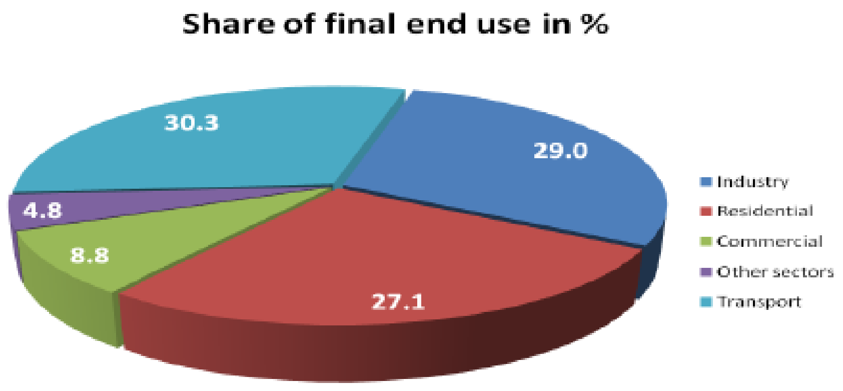


Figure 3 Energy consumption in different sectors [13]

Lately, more and more solutions and technologies are attempting to combat this by reducing the greenhouse gas emissions either by changing the way we travel or by completely changing the energy source. Electric cars are on the rise and countries are trying to move from fossil fuels to other more environmentally friendly sources to produce electricity to power the transportation in the cities.

But changing to an electric vehicle is not yet a complete solution for combating emissions, while they are considered to contribute to the reduction of greenhouse gas emissions and local air pollution, the energy used may still come mainly from fossil fuel burning. [14]

It is argued that, from society's point of view, cities need to consider other mobility options, such as metros, trams or electric buses, while promoting walking and biking for its citizens. [14]

2.2. Green transportation

Worldwide, in 1990 using motor vehicles for transportation lead to 14% of carbon dioxide emissions resulted from fuel burning [15]. In the 2000s, the whole transportation sector was responsible for over 30% of global emissions [13]. In the context of human driven climate change due to greenhouse gas emissions, we have the responsibility of solving this issue. It shouldn't fall just on governments or corporations, each individual needs to consider what their impact is.

Different transportation methods employed by each individual impact differently the environment. Table 1 shows the Environmental Friendliness (EF) for each transportation type, and what are the discouraging and encouraging factors for each. The goal for individuals is to maximize their EF. Approaches include walking, cycling and using public transport, by making it more attractive, or optimising and limiting the use of private vehicles. Nudging may be a great method to help them achieve this. [16]

Table 1 Environmental friendliness (EF) and discouraging and encouraging factors for different types of transportations. Larger EF means more environmental friendly type of transportation [16]

Type	EF	Discouraging factors	Encouraging factors
Car	🟩⬜⬜	Economy (toll, parking, gas), traffic jam	Convenience
Carpool	🟩⬜⬜	Inconvenience, traffic jam	Economy, social
Bus	🟩🟩⬜	Schedule, traffic jam	Economy, priority in traffic
Bike	🟩🟩🟩	Time, effort, exposed to conditions	Economy, health
Walk	🟩🟩🟩	Time, effort, exposed to conditions	Economy, health

There are already some implementations for systems that try to help people reduce their carbon footprint, or in other words, their impact on the environment. Some help people by providing them with a carpool sharing system [17], where people that have similar destinations can share cars. Encouraging walking and cycling, or promoting public transportation, instead of increasing road space, an impulse for people to use their private car leading to maintaining the congestion problem [18]. Some results show that a more efficient public transportation leads to a lesser use of private cars, bus priority measures being applied in urban areas [19]. It's been suggested that the access to urban areas should be restricted when pollution or congestion reach maximum limits or setting a toll for entering zones with the greatest traffic pressure, such as the city centre or the business district [18]. There are such measures being employed in Spain, both in [20] Barcelona and Madrid [21], where the authorities limit the number of cars allowed in the city in the case of high pollution episodes.

2.3. Nudging

Nudging, as defined by Thaler and Sunstein, is a way of influencing people's decisions and behaviour by making subtle, seemingly insignificant changes of the decision making context [22]. It's a concept in behavioural science, political theory and economics which proposes positive reinforcement and indirect suggestions as ways to influence the behaviour and decision making of groups or individuals.

A nudge is not a mandate, its intervention must be easy and cheap to avoid [23]. Prohibiting alcohol or smoking, for example, are not considered nudges, as it moves the decision-making context away from the person. Also, raising the prices of alcohol and cigarettes is not a nudge, as the intervention becomes expensive to avoid for the user, but, otherwise, putting pictures on the packet with the effects of smoking is a nudge.

Nudges are considered a form of libertarian paternalism. A policy is paternalistic when the actor aims to benefit a person, even against his will [24], while an action is libertarian paternalistic when the option of choosing isn't stripped away from him, so "if it tries to influence choices in a way that will make choosers better off, as judged by themselves" [23]. Based on this idea, nudges are acceptable based on welfare premises, they help the individuals that are being nudged or the society in general.

Furthermore, it is argued that nudges and choice architectures are unavoidable and, on some occasions, required on certain ethical grounds. [25] We are nudged unknowingly almost all the time, whether we are in a store or voting for representatives. A choice architecture represents simply the design where choices are presented to users, which, willingly or unwillingly, influences the decision making of the users.

The nudge concept is an interesting theory, that's why several experiments were run to see how the theory reacts in different real-life scenarios. Many methods were employed in different areas.

- An experiment ran in a school cafeteria showed that if healthier foods were displayed at the shopping queue, the students had a healthier food intake than otherwise. The results showed that students increased their intake of healthy foods by 18% and decreased the intake of less healthy ones by 28%. [26]
- An airline company's main operation cost is represented by fuel consumption (which is about one third of the total operational costs) but, despite knowing that they should reduce fuel consumption for both lowering costs and carbon dioxide emissions, pilots don't usually act accordingly. One airline company tested a few nudges to see which would offer benefits. In their experiment, they split the pilots in 4 groups: a control group, which got notified that their fuel consumption will be monitored as part of a study and 3 other groups which received interventions during the experiment: either personalised reports, fuel consumption goals or a small donation in their name if they successfully reduced their fuel use. By the end of the experiment all groups successfully modified their behaviour for the better, even the control group [27]. This shows that something as simple as informing a person can end up nudging him, changing his behaviour, by making him more aware of his actions.
- Farmers in Kenya didn't take advantage in profitable fertilizer investments, but responded positively to small, time-limited discounts of acquiring fertiliser just after harvest. The researchers suggest that this can yield even higher welfare than heavily subsidising them. [28]

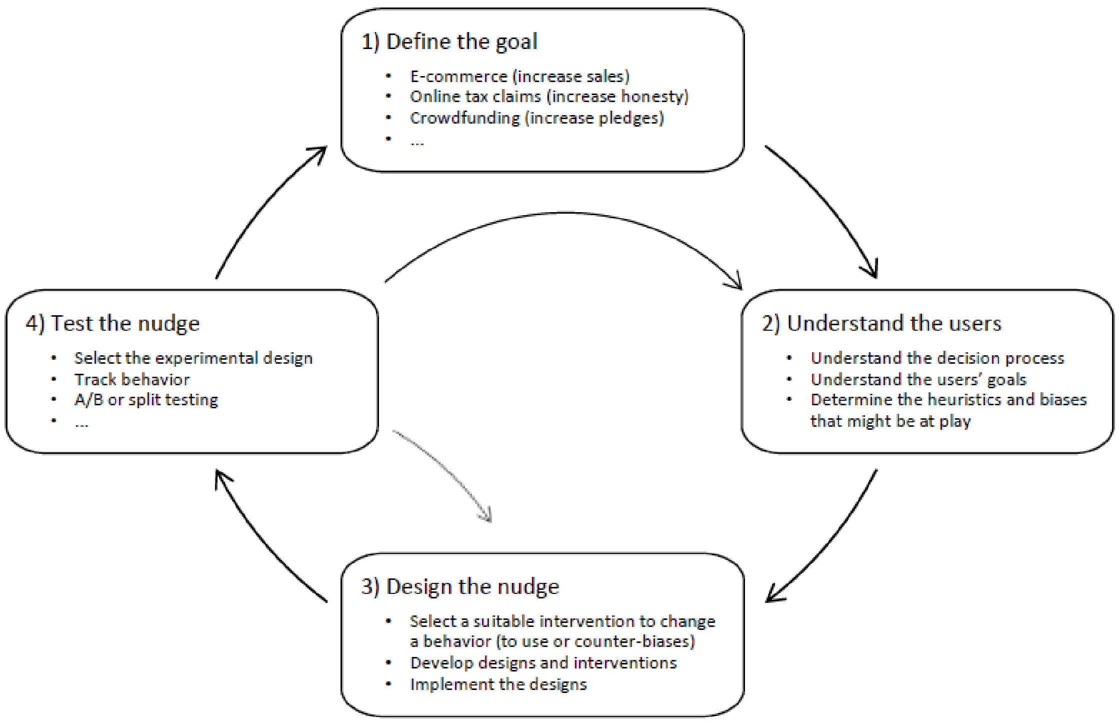
A nudge involves two main groups of people: the choice architects (people who design and are issuing the nudges) and the people who are being nudged. Despite being stated that nudges are considered to be libertarian paternalistic, there have been voices who emphasize the fact that nudge theory can be used to manipulate people for egoistic reasons. For this reason, transparency is a great asset to be employed in creating nudges. The people being nudged should know when, how and why they are nudged. [25]

Personalization is very important when designing nudges, one method that works on a person or a group may not work on another. For successful nudging, the choice architect needs to understand the users he is designing the nudge for. One method is studying the users beforehand, designing the nudge and, depending on the results, continuously redesigning the nudge.

Nudges may use familiar technologies: a push notification or even an email may be considered a nudge. Digital nudging refers to using nudges in the interfaces of software applications, in other words, the use of the user interface to influence people's behaviour in digital choice environments. [29]

In context of digital nudging, the software developer building an application is, willingly or unwillingly, a choice architect, because he is the one defining the context in which the user makes his choices. In order to know how to do this better, there are some defined guidelines that a choice architect may choose to follow, similar to traditional nudges.

The steps for designing digital nudges are represented in figure 4.



1) Define the goal
E-commerce (increase sales)
Online tax claims (increase honesty)
Crowdfunding (increase pledges)
...
2) Understand the users
Understand the decision process
Understand the users' goals
Determine the heuristics and biases that might be at play
3) Design the nudge
Select a suitable intervention to change a behavior (to use or counter-biases)
Develop designs and interventions
Implement the designs
4) Test the nudge
Select the experimental design
Track behavior
A/B or split testing
...

Figure 4 Designing digital nudges follows a cycle [30]

The choice architect needs to first define his goal: what is he nudging for or what does he want the users to do. In this step, the architect needs to consider the ethical implications of the nudge, as nudging people into making detrimental decisions of their wellbeing can have negative effects on the organisation he is working for. The types of nudge approaches vary based on the goals. The next step is to understand the users: different user group will respond differently to nudges, so it's important to understand how they can be influenced and what goals the users have. After these 2 steps, the nudge is designed. Here is where the choice architect designs the choice context or creates a suitable intervention method to influence the users. After this step, the nudge is tested, and if unsuccessful, the nudge design needs to be changed directly, or the choice architect needs to better understand the users before redesigning the nudge. [30]

Similar to traditional nudges, digital nudges have been previously employed with successful results:

- Square is a payment app that presents a tipping option as default, so users need to select the "no tipping" option if they decide not to tip. The application promotes tipping since the user will usually choose the default option. [29]
- In a series of experiments, in context of reward-based crowdfunding, researchers have discovered some methods to influence backers to pledge more money to a project. The decoy effect shows that adding an option, which no one will reasonably choose, alongside other ones,

will increase the attractiveness of it. Also, the scarcity effect shows that users tend to perceive scarce items more desirable than common ones, for example when limiting the availability of rewards. The middle-option bias shows that people tend to choose the middle option more. The experiment showed that people preferred the middle option no matter if the options of paying were {$5, $10, $15}, {$10, $15, $20} or {$15, $20, $25}. [31]

- A team developed and tested a nudging solution to stop procrastination in classrooms. They used text messages to send personalised automated nudges to college students. To achieve this, the researchers developed a chat bot that ask students multiple choice questions with randomized answers. The bot was used to encourage learning. Students were checking their knowledge against the bot's questions and in case of a wrong answer, they would go back to study. The results showed that, overall, students that used the chat bot ended up with better grades at the exam. [32]
- By displaying the strength of a password, users are more likely to choose a stronger one, helping in security and privacy. [29]
- A step counter app that provides feedback can push users to be more active by setting small goals throughout the day. [29]
- Smart meters are used to encourage energy savings, reducing greenhouse emissions. [29]

2.4. Data management and Big Data

Data management is the practice of organising and maintaining data processes to meet ongoing life cycle needs. It's the intelligent use of scarce resources to enhance data access [33]. It is very important to consider what options there are for storing and handling data, considering from the start what data we are going to handle and what kind of technology is best suited for our needs.

The traditional way of storing data is using a relational database, which is based on the relational model. This is a method that uses a collection of tables to represent both data and the relationships between data. Such systems are named Relational Database Management Systems (RDMS). The most common way to manipulate data in a RDMS is using Structured Query Language (SQL), which is used for querying, inserting, altering or deleting data [34]. For this reason, relational databases are often called SQL databases. Having relationships between tables greatly simplifies the maintenance of the data and ensures its integrity.

Relational databases inherently support ACID transactions (atomicity, consistency, isolation, durability) [35], which greatly simplifies the user's work. Transaction processing guarantees a consistent and reliable execution of applications in the presence of concurrency and failures [34]. A database transaction is a series of operations that changes the database from one consistent state to another. The ACID properties are resulted from always keeping the database in consistent states [36]:

- Atomicity – a state transition appears to jump from the initial state to the resulted one, without any intermediate steps (all-or-nothing principle), in other words, either the transaction completes successfully, or it's not executed at all.
- Consistency – a transaction either produces only consistent results, or it aborts.

- Isolation – concurrent execution of transaction leaves the database in the same state as it would have executed sequentially.
- Durability – once a transaction is committed, it will remain committed even in the case of system failure.

The main issues encountered by RDMS when working with big data sets are: scaling out the data, maintaining performance on single servers and having a rigid schema design. [37]

One of the greatest challenges of relational databases is scalability. One way to classify scalability is: vertical scalability, when adding more computational power by improving the software or the hardware resources, deploying on large shared-memory servers; and horizontal scalability, when adding more of the same software or hardware resources, deploying on distributed servers [38].

Originally, RDMS were not built with horizontal scaling in mind. Vertical scalability proves costly and impractical because of hardware limitations, while scaling horizontally by spreading among many server nodes proves hard on relational databases. For the same reasons they are not well suited for working in the cloud either [39], as described later.

When talking about how to manage data, we need to take in consideration the term Big Data and what it involves, and especially to look at how we can handle it. RDMS has proven in the past that it is hard for it to handle this kind of huge data sets.

Big data is data that can be defined by some combination of the following 5 characteristics [40]:

- Volume – the amount of data to be stored and analysed is big enough that will require special considerations;
- Variety – the data consists of multiple types, maybe from multiple sources. Structured, semi-structured and unstructured data needs to be considered;
- Velocity – the data is produced at fast rates and old data may not be valuable;
- Value – the data has perceived or quantifiable benefit to someone who uses it, an enterprise or an organisation;
- Veracity – the correctness of the data can be assessed.

Companies that are involved with Big Data and internet services are growing rapidly. For example, Google processes hundreds of Petabytes (PB) of data and Facebook generates tens of PB per month. The amount of large datasets is rising rapidly, bringing along other problems, which require immediate attention [41]:

- The latest advancements in information technology makes it easier to generate data, therefore there is a challenge in collecting and integrating data;
- The rapid growth of cloud computing and Internet of Things promote sharp increase of data; this implies that there are problems in storing and manage such heterogeneous datasets;
- Considering the heterogeneity, scalability, real-time, complexity and privacy of Big Data, there are methods to "mine" the datasets at different levels to consider, analyse, model, visualise and forecast.

It is required special focus when working with the sheer amount of data that needs to be collected and processed when Big Data is involved.

Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction [42]. It seems to be able to tackle the issues coming with big data. The term *cloud computing* denotes the computing infrastructure as a "cloud" from which users are able to access applications, on demand from anywhere in the world [43].

To process and store the massive amounts of data more easily, a strategy is to partition the data across different server nodes, which in turn can be replicated in multiples servers, for the data to be available in case of server failures [44]. The problem of such systems is formulated by the CAP theorem: only 2 of the 3 CAP properties (consistency, availability, partition tolerance) can be satisfied on any given time [45]. In other words, systems need to choose availability versus consistency in the face of network partitioning. Consistency refers to having a single up-to-date instance of the data, availability tells us that the data should be able to serve requests whenever necessary, while the partition tolerance refers to the capacity of the shared system to tolerate network partitions.

The emergence of cloud computing makes developing applications easier for the development team, as they don't need to handle the computational or the data servers, this being handled by third parties (ex.: Microsoft, Amazon, etc.), which will add new nodes, or remove others if necessary.

Challenges of handling big data and use of distributed systems techniques in the context of CAP theorem led the development of data sores called NoSQL and NewSQL [44].

NoSQL, acronym for "Not only SQL", is an umbrella term of diverse data stores that are not based on the relational model. A NoSQL solution usually has the following characteristics attributed to them [46]:

- It has a simple and flexible non-relational data model. They offer flexible schemas, or they are completely schema free, being designed to handle a wide variety of data structures.
- It scales horizontally very well over many commodity servers, either by offering data scaling or read/write scaling.
- It provides high availability, by compromising consistency in case of network partitioning.
- It doesn't support ACID transactions, using BASE instead.

While ACID provides the consistency choice for partitioned databases, BASE offers availability instead. BASE stands for [47]:

- Basically Available – the database system always seems to work, it guarantees availability based on the CAP theorem. Such systems spread data across many storage systems with a high degree of replication.
- Soft state – the copies of the data item may be inconsistent, and the state of the system may change over time, even without input, as an effect of eventual consistency

- Eventually consistent – the updates propagate through the system, which becomes consistent if there are no more updates over a certain period of time.

NoSQL databases mainly differ from relational databases in their data model. There are mainly 4 groups of NoSQL databases [48]:

a) Key Value Stores
Key values stores are completely schema free. They are similar to maps or dictionaries, as they are addressed by a unique key. The values are represented in byte arrays, which are completely opaque to the system. Values are isolated and independent, relationships being solved in the application level. An example of this is Redis.
b) Document Stores
Document stores are key value stores that encapsulate the pairs in JSON, or JSON like documents. Within the documents, keys must be unique, while each document has a special unique key that identifies it. In contrast to key value stores, values are not opaque to the system and can be queried. MongoDB is one of such stores.
c) Column family stores
In column family stores, the data model is described as a "sparse, distributed, persistent multidimensional sorted map" [49]. In this map, an arbitrary number of key value pairs can be stored within rows. Similar to key value stores, relationships need to be implemented into the application logic. Columns can be grouped into column families to help with partitioning and data organisation. They also support versioning; each version of a value is stored in a chronological order. Google's Big Table is one of the most prolific examples.
d) Graph databases
Graph databases are specialized on efficient management of heavily linked data. Data with many relationships are well suited for these kinds of databases, because recursive joins can be replaced with efficient traversals. Nodes and edges are a representation of objects with embedded key-value pairs. Twitter's FlockDB is an example.

While there are benefits for the NoSQL databases, the downsides can't be ignored. Because of no consistency guarantees offered by ACID transactions, developers must enforce consistency on the application level. Also, the lack of adoption of high-level query languages with built in optimizations, means that the application developer needs to worry about optimizing the execution of his data accessing methods. Changing from one NoSQL system to another proves difficult because there are no standards employed for APIs for data access and manipulations. It seems like too much burden is put on the application developers and database administrators, while the extent history and evolution of RDMS with its lessons learned are mostly ignored. [50]

The lack of ACID transactions and the huge investment in SQL by enterprises are barriers in face of adoption of NoSQL data stores. The category of NewSQL data stores classifies a set of solutions aimed to bring the relational model to the benefits of horizontal scalability and fault tolerance offered by NoSQL solutions [44]. NewSQL is a class of modern relational DBMS that seek to provide scalable performance of NoSQL for OLTP (On-line Transaction Processing) read-write workloads, while still maintaining ACID guarantees for transactions [51].

FaRM (Fast Remote Memory) is a main memory distributed computing platform developed by Microsoft, which can provide distributed ACID transactions with strict serializability, high availability, durability and high performance. [52] They managed to achieve this with 2 hardware trends available in data centres: fast commodity networks with RDMA (Remote Direct Memory Access) and non-volatile DRAM (Dynamic Random-Access Memory), achieving non-volatility by using batteries to write the contents of DRAM on SSDs in case of power fails.

For long term storage of data that isn't accessed that often, using a data warehouse is a better approach than using OLTP solutions. A data warehouse is a subject-oriented, integrated, time-variant, and non-volatile collection of data in support of management decision-making process. It uses OLAP (On-line Analytical Processing) instead of OLTP [53]. It is especially used for storing data used in data analysis, as it stores data that isn't volatile, the data sets only grow, offering a view of the data over time.

2.5. Approximate Query Processing

A method of processing big data sets is using Approximate Query Processing (AQP), which offers the possibility of getting approximate answers at a fraction of the computational cost of doing the query in the traditional way. An AQP scheme may be characterised by the generality of the language it supports, it's error and the accuracy model, the amount of work saved at runtime, and the amount of additional resources needed for the computation. [54]

There are 2 methods of achieving AQP: either using query time sampling, when the user specifies sampler operations in the query, or by drawing samples from the data in a pre-processing step, which can yield even faster results. [54]

Using AQP techniques may produce valuable results in a shorter amount of time than using normal queries, especially when querying huge amount of data. A query can significantly reduce its response time, while still having a rough, approximate, result, with a defined error range.

2.6. Key-Value In-memory Data Stores

Key-value stores are schema free data stores with a simple model, storing a single key-value index for all data [55]. The data store provides inserts, deletes and lookups based on keys, with other functionalities based on implementations. The values stored are byte-arrays, opaque to the system, with no relationships between them, which are addressed by unique keys. Relationships may be defined in the application level, if necessary. [48]

Redis is an example of such data store. It also features an expiration functionality, key-value pairs having the possibility of being provided with a lifetime or an expiration date, Redis automatically deleting the pair when the expiration time arrives. This is useful for keeping the memory usage low and cleaning the data that is no longer needed. [56]

2.7. Coding patterns

When building an application, it is important to have in mind some patterns in order to achieve a readable and maintainable code base, while keeping it open for later extensions.

The repository and unit of work patterns add a thin abstraction layer on top the data access of an application. Figure 5 show an example of how this looks.

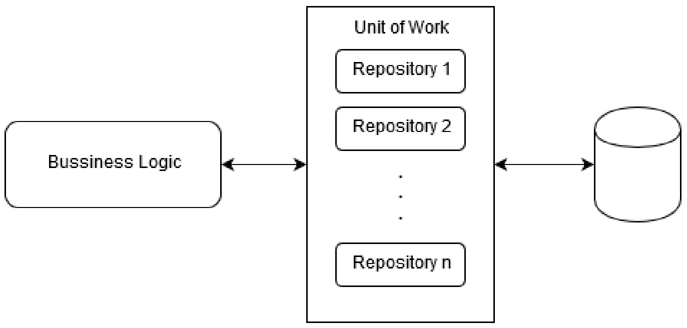


Figure 5 Repository and Unit of Work

The repository pattern implements repositories, which are classes that encapsulate the logic required to access data structures, centralizing common data access functionality [57]. It removes duplicate code, centralizes data-related policies in an app and helps with testing the business logic, isolating from external dependencies. [58]

A unit of work maintains a list of objects affected by a business transaction and coordinates the writing. The idea is that, instead of doing a lot of small database calls, all the transactions are coordinated and all the changes to the database are done in one database call. [59]

The Dependency Injection pattern is a technique of passing dependencies to a dependency injector, which handles the creation and initialization of dependent objects [60]. This kind of behaviour enables loose coupling, which reduces the risk of changing one element creating unanticipated effects in others. Also, using a dependency injection technique is helpful in unit testing, offering the possibility for the software developer to inject mocked objects into the test instead of real ones. The disadvantage of this pattern is that, if done incorrectly, many compile errors are moved to run-time.

There are 3 types of dependency injection:

a) Constructor injection – when the dependencies are provided through the class constructor.
b) Setter injection – when the object exposes a setter method that is used to inject the dependency.
c) Interface injection – when the object provides an injector method, exposed in the interface, that will accept the dependency.

The concept behind dependency injection is inversion of control [61], by which the control of objects in the application is transferred to a container or framework.

2.8. Related work

This section will focus on other digital nudging solutions, similar with ours, that use nudges either to promote health and mental wellbeing, or green transportation choices.

Mobility choices app

The application offers the possibility for the user to create a profile with his preferences based on health, environment, costs and preferred transport. This may include means of transport, wait time, number of changes, and walk time. All the data is used to provide the user with the possible greener modes of transport. [62]

UbiGreen

The application tracks the user's travel method using external sensors and provides active feedback of their impact on the environment. The feedback is presented on the user's screen either using a tree getting greener as they were using more environmentally friendly modes of transport, or an eco-system with polar bears, fish and seals, which got richer as the user travelled greener. Failure is represented by either a sparser tree or by thinning the ice and fewer animals. The researchers' focus was on the immediate feedback to the user. [63]

A better day

The application works on improving sustainability for nutrition, mobility and living. The idea is using a 100-point daily budget, which represents the amount of CO_2 a person can emit daily, which the planet is capable to replenish. The app makes it easier for users to change their lifestyles, by assessing the costs of everyday life and creating challenges and guides to make users simplify their life. [64]

E-Nudging - motivational aid in the prevention and treatment of chronic diseases in everyday life

The goal of this application is to use digital nudging in prevention and treatment of chronic diseases. The researchers throw light on the pros and cons of digital nudging and they emphasize the design concepts to consider. They highlight the importance of design simplicity and setting small goals for the users, easier to achieve, rather than discouraging them using hard or unreachable ones. The main problem they encountered is keeping the users motivated. [65]

Summary

The mentioned applications have an approach either in nudging, transportation or both. The mobility choices app has a preferences method that we partly employed. As suggested in the E-Nudging app, we adopt simplicity in our nudges, with simple goals, easier to achieve.

A better day has a challenge approach to make users change their default lifestyle choices, while it's an interesting approach, we decided to use a simple notification system. UbiGreen has an interesting approach for inferring the user's mode of transport using sensors, which would be useful in our case. Even so, we chose to rely on direct user feedback.

As in the example of the flight company presented in the Nudging section, it’s enough to notify people that they need to make a change, and change will happen. We rely especially on this effect of nudging. Also, as suggested in the Square tipping application example, users rely heavily on default options. We are able to use this to our advantage, nudging people on more varied premises, while the people who aren’t affected by nudges or are notified too much, can modify their nudging preferences.

3. Architecture and Design

This section describes the approach in solving the challenges of nudging for transportation, the general architecture of the application, approach to the data management and the database design.

3.1. General Approach

The issue of nudging for green transportation involves multiple challenges, the main of it being convincing people of using a more environmentally friendly mode of transport, choosing more Environmental Friendliness, as described in section 2.2: each mode of transport has encouraging and discouraging factors, our point is to convince users that encouraging factors overweight the discouraging ones. This involves nudging people for changing their default mode of transport for more environmentally friendly one, generally to convince them not to use their personal car.

Building an application that achieves this, represents a challenge, as it needs to adapt its nudges in regard to the people's preferences and how they react. We can assume that users of the application want to change, as they chose to use the application, but changing their default transport mode takes more than just sending them simple messages.

Our main approach is to build an application that collects data from multiple resources and, based on user activity, decide to nudge under different situations. To achieve this, we need to provide the application with environmental data from different sources, current and historical, and track how the users respond to different kinds of nudges depending on the situation.

We decided for the application to be divided into two main systems: the data analysis service and the data management service, each with its own important responsibilities. This design choice is made to separate the main challenges of the application.

The Data Analysis Service handles the decision making in the application. It runs in the background, triggered by user or public events, analyses the data and decides if it is necessary to nudge the user or not. It requires user historical data, represented by how user reacted to nudges under certain situations, data about other users and current data about the environment. This is explained more in [1].

The Data Management Service handles the data storage and access. It handles heterogeneous data from multiple sources, filtering and storing what is needed. It also offers fast access to the data for analysis. This is the main focus of this thesis.

3.2. Application Architecture

The application architecture represents all the services and modules that run in the application. In this section there is a short description of what each service does and how they communicate with each other.

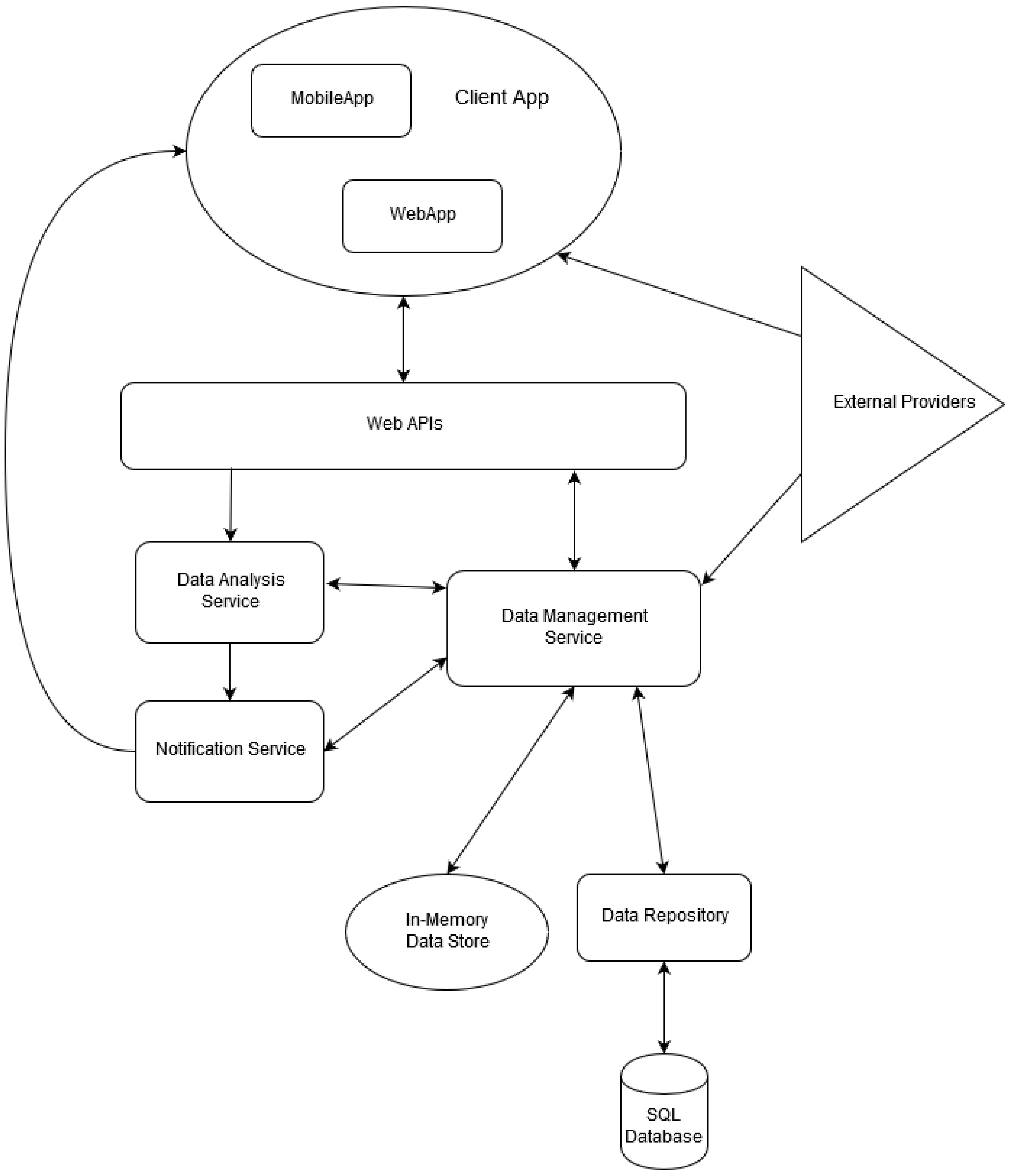


Figure 6 Application architecture

Figure 6 shows a representation of the application architecture, with all the component parts and relationships. The arrows represent application triggers and how the data travels. All the parts of the application can be separated in a few different modules: The Client App (Mobile App and Web App), the business logic (Data Analysis Service, Data Management Service, Notification Service), data access and storage (Data repositories, databases, data store) and external providers.

ClientApp

The Client App represents the software running on the client side, either a web application or a phone app.

The web page is designed as a single page application (SPA). The benefit of this, as opposed to traditional web applications, is that it can dynamically update the content without triggering a page reload. That means that the server communicates continuously with the web application and the user.

The Client App communicates with the server application through the WebApi, asking for or saving data, or triggering the Data Analysis Service. I also can directly request data from the external providers, bypassing our server. It can receive nudge notifications from the Notification Service.

WebApi

The Web API layer serves all the API requests coming from the ClientApp and forwards them to the required service. This service runs on the server and acts like an entry point for all requests. It represents Controllers which handle the requests, convert data into something the system can understand and forward those requests to the appropriate services.

The Client App makes a http request towards the server, the controllers pick up that request, authorize it if necessary, and then forwards the call to either the Data Analysis or Data Management Service. When the services handle the request, the controller formats the data and sends it back to the Client App if necessary. If the request from the Client App doesn’t expect any data to be returned, then it would expect just an Ok response once the request finishes.

Data Analysis Service

It’s the service that does analysis on the user data to trigger a nudge notification. The analysis service gets all the data from the Data Management Service, which means that it doesn’t care whether the data is stored or it’s live data, or where the storage of the data is handled.

This service runs automatically in the background, running time based on a different process, or can be triggered externally, by the Client App. Once triggered, it gets the data and decides if a notification should be sent or not, through the Notification Service.

Notification service

The main purpose of the notification service is to handle sending notifications to users. It receives requests from the Data Analysis Service and sends push notifications to the Client App. It uses the Data Management Service to access the endpoints where the notifications should be sent for each user.

Data Management Service

This is the service that handles saving, converting and serving the data to the Web App, the Analysis Service or the Notification Service. It either accesses the Repository Service for persistent data, the In-Memory Data Store for transient data, or fetches data from the external providers for getting live data.

All data operations pass through this service. It knows which services needs what data and handles all the conversion. It offers one extra layer over the storage system, hiding access methods and ensuring fast access, if the data is persistently stored or provided from external sources.

Data Repository

This service handles the actual connection and communication with the database. It holds all the database objects and has the appropriate methods for storing or querying the database. It masks everything related to the database, hiding how the communication is done. This helps in maintenance by separating the code in charge of the database connection from the one with the business logic. It also simplifies the work in case of database change. This is important as seen in the future work, due to the design choices made in the implementation.

Having this layer also helps in unit testing the business logic, because it makes it easier to create a mock of the storage layer.

It represents an approach to the Repository coding pattern presented in the background at section 2.7.

SQL Database

The SQL Database is a relational database used for persistent storing of user information, preferences and nudge data, which includes trip and environmental information. It also holds data about notifications, where notifications are sent and notification history. It has approximate query processing capabilities, ensuring fast query response times on huge data sets.

In-Memory Data Store

The In-Memory Data Store is used for in memory caching data not required for permanent storage. It is used as a Key-Value data store. The data is stored for a certain amount of time for efficiency reasons. If multiple users access the same external data resources in a certain time frame, we serve the data from memory, rather than externally requesting it.

It stores data in Key-Value form, with easy generatable keys and without using any data schema, as presented in section 2.6 of the background. The motivation of using such a store is for simplicity and fast data access, offering the possibility for storing some data for short amount of times, that otherwise would be accessed from external providers.

External Providers

The external providers imply all the systems we receive data from, that are not maintained by us. They provide the weather forecast, traffic information, the location, travel directions, bus schedule and calendar events.

The providers are either accessed from the Data Management Service or directly from the Client App, to avoid some privacy issues and latencies.

3.3. Data Management Approach

My approach to solve the challenge of handling multiple heterogeneous data is creating a special service inside the application than handles all the data storage and access, keeping the methods separate from other services in the system. This service is called the Data Management Service. It handles storage, persistent and transient, requesting data from external providers, data conversion and data access.

I refer to transient data, as in temporary, volatile data, that doesn't need persistent storage. It is stored for a short amount of time, in general for improving response times.

To handle slow response times from external providers, data is stored in memory, caching certain types of data that changes in short amounts of time. If multiple requests are coming in the lifetime of the stored data, the requests are served from memory rather than from the external resources. The data is stored in a Key-Value form with easy to generate keys.

The heterogeneous data management challenge, generated by all the different external data sources, is solved by lightly processing the data to convert it into an aggregated form at the moment it is received. The service gets the data from external resources, strips away what is not necessary and combines other values to get more valuable information, before storing it either in memory or in persistent storage, as necessary.

Persistent storage is handled using a SQL database with AQP capabilities, so it can handle certain query types faster, while providing valuable approximate answers. Approximate Query Processing, as explained in section 2.5, is a technique of providing approximate results with a defined error in a shorter amount of time that is a fraction of the time an exact result would be provided.

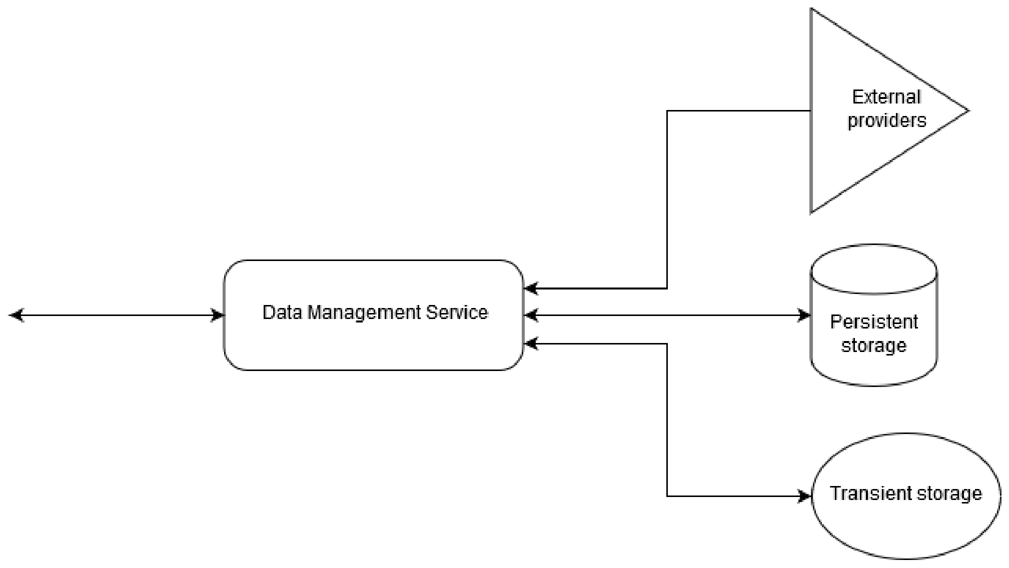


Figure 7 Data Management Service

3.4. Database Architecture

The user and nudge data are persistently stored in a relational SQL database with 7 tables. Figure 8 contains an overview of the tables and relationships between them. The tables hold user information, such as email and password, user preferences, notification information and the nudge history, which contains information about trips, environmental information and nudge results.

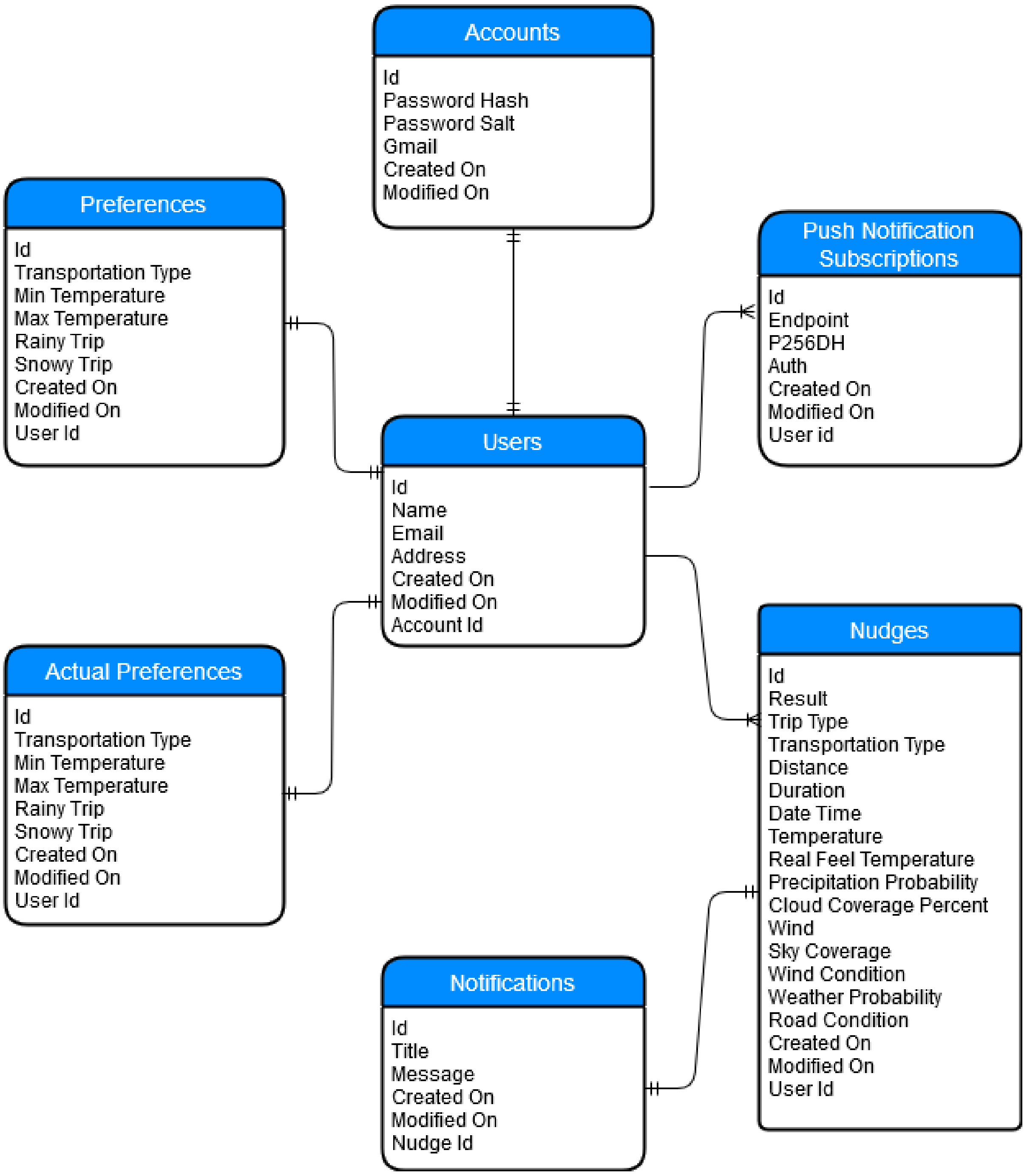


Figure 8 Database Architecture

All tables contain 2 extra columns besides the Id (the unique identifier of the row): *Created On* and *Modified On* columns, which contain the time and date when the entry was inserted into the table and, respectively, when it was last modified. This information is mainly useful when auditing the data or in debugging, in case of failures in the code, and in some cases it is displayed to the user. Each table holds specific user or nudge information:

Users

The Users Table mainly holds user information: name, email and address. We have a one to one relationship to the Accounts table. The email is unique and mandatory, as we identify the user by it.

The table contains a foreign key to the Account Table.

Table 2 Users

COLUMN	DATA TYPE	NULLABLE
NAME	Text	Yes
EMAIL	Text	No
ADDRESS	Text	Yes
ACCOUNT ID	Unique identifier	No

Accounts

The Accounts Table holds user credentials data and marks if the user logged in with Google. We decided to separate this, as the data is necessary only in the authentication and registration phases.

The user can authenticate with local login, providing an email and password, from which a password hash and salt is produced and stored. The salt is a randomized string used for hashing the password.

If the user opts to use Google log in, the authentication is not performed in our application, as we trust Google to do it for us, so no password information is stored in our database.

Table 3 Accounts

COLUMN	DATA TYPE	NULLABLE	DEFAULT
PASSWORD HASH	Text	Yes	Null
PASSWORD SALT	Text	Yes	Null
GOOGLE	Bit	No	False

Preferences

The Preferences Table holds user preferences data. It contains the preferred transportation type, the range of temperature where he is fine traveling, and if he is fine with traveling when raining or snowing. The data in this table is inserted by the user.

It holds the foreign key to the associated user in the user table.

The transportation type may be: walk, cycle, bus, car or unknown, each with an associated number.

As a default the temperature is set to the extremes, so it won't influence analysis, the transportation type is unknown, and the snow and wind trip are enabled. In these cases, the preferences should not be considered.

Table 4 Preferences

COLUMN	DATA TYPE	DEFAULT
TRANSPORTATION TYPE	Number	Unknown
MIN TEMPERATURE	Number	-50
MAX TEMPERATURE	Number	50
RAINY TRIP	Bit	True
SNOWY TRIP	Bit	True
USER ID	Unique identifier	-

Actual Preferences

The Actual Preferences Table is similar to the Preferences Table, the difference is that the data here is generated based on user activity by analysing the nudge acceptance.

It holds the foreign key to the associated user in the User table.

Table 5 Actual Preferences

COLUMN	DATA TYPE	DEFAULT
TRANSPORTATION TYPE	Number	Unknown
MIN TEMPERATURE	Number	-50
MAX TEMPERATURE	Number	50
RAINY TRIP	Bit	True
SNOWY TRIP	Bit	True
USER ID	Unique identifier	-

Push Notification Subscriptions

The Push Notification Subscriptions table holds information about where to send the notifications, so they can reach the user. When a user accepts the push notifications feature, we are provided with an endpoint to which to send and some authentication credentials to attach with the notification.

The endpoint represents where the notification is sent, this is represented by the running Client App (ex.: the browser), and the P256DH and AUTH are used for authorization with the endpoint. The process will be explained in the Implementation part of the paper.

Each user may have multiple endpoints for receiving notifications, as he can use multiple devices to log into our system, so we will have a one to many relationships between the User and the Push Notification Subscriptions tables. The table holds the User Id foreign key.

Table 6 Push Notification Subscriptions

COLUMN	DATA TYPE	NULLABLE
ENDPOINT	Text	No
P256DH	Text	No
AUTH	Text	No
USER ID	Unique Id	No

Notifications

The Notifications Table holds the actual text notification for each nudge. Each notification entry is linked to a nudge in the Nudges table. Some nudges may not have any notifications associated with them, as they are created in other conditions, without triggering any notifications to the user.

The data is presented to the user as a history of all nudges.

Table 7 Notifications

COLUMN	DATA TYPE	NULLABLE
TITLE	Text	No
MESSAGE	Text	No
NUDGE ID	Unique Id	No

Nudges

The Nudges Table contains all the information regarding nudge results, traveling information and weather forecast for a specific trip of a user.

We store if the nudge was accepted or not, or if this information is unknown, as the nudge Result. It also holds trip distance, duration and transportation type. Each trip has a type, to define what kind of information we know about it: a walk or a trip with destination. If the user is nudged for a leisure walk we don't know how long the trip was or the distance travelled, as the user has no actual destination, but information about the trip may be still valuable for future nudging.

The table contains also raw weather data: temperature, real feel temperature, cloud coverage and wind speed; but it holds aggregated data as well: the sky coverage (clear, cloudy, partly cloudy), wind condition (strong winds, light winds, calm), road condition (dry, wet, ice, snow) and weather condition (none, rain, snow, no snow, freezing, cold, cool, warm).

Having all this information in one table helps simplifying the query and avoiding table joins for faster results.

This table has a one-to-one relationship to the user table.

Table 8 Nudges

COLUMN	DATA TYPE	NULLABLE
RESULT	Number	No
TRIP TYPE	Number	No
TRANSPORTATION TYPE	Number	No
DISTANCE	Number	Yes
DURATION	Time	Yes
DATE TIME	Time	No
TEMPERATURE	Number	No
REAL FEEL TEMPERATURE	Number	No
CLOUD COVERAGE	Number	No
WIND	Number	No
SKY COVERAGE	Number	No
WIND CONDITION	Number	No
WEATHER PROBABILITY	Number	No
ROAD CONDITION	Number	No
USER ID	Unique Id	No

4. Implementation

The implementation is an example of the approach: an application involving nudging for green transportation and my proposed solution for data management for such an application. The proposed implementation is specifically created for the area in and around Tromsø, further modifications are needed for a working solution in other areas.

This chapter focuses on what tools and technologies we chose to use and then dives into the implementation of the data management service and of the application in general.

4.1. Tools and technology

This section explains what technologies we chose and why and goes through the tools used for development.

4.1.1. *Backend*

Microsoft Visual Studio Community & C#

Microsoft Visual Studio is an integrated development environment developed by Microsoft [66]. We used the free version of it: Visual Studio 2017 Community.

C# is a general-purpose, multi-paradigm programming language encompassing strong typing, lexically scoped, imperative, declarative, functional, generic, object-oriented, and component-oriented programming disciplines. [67]

The combination of Microsoft Visual Studio and C# was chosen for the versatility of the technologies and for the ease of use, as I am experienced in implementing applications using them.

For downloading third party dependencies we are using NuGet package manager, which is integrated in Visual Studio [68]. Some of such dependencies used are *RestSharp* [69], *WebPush* [70], etc.

.Net Core Framework

.Net Core is a general-purpose development platform maintained by Microsoft [71]. It is cross-platform and can be used to build device, cloud and IoT applications.

We chose the .Net Core 2.2 Framework for its versatility, as it runs on all platforms: Windows and Unix systems, both Linux and Mac.

Entity Framework Core

Entity Framework Core (EF Core) is a lightweight, extensible and cross-platform data access technology. [72]

EF Core implements the Repository and the Unit of Work Patterns, as explained in section 2.7. It handles the connection with the database and allows working with the database context, where you can manipulate to all entities and then do a commit, so that all changes are done in one operation on the database (implementing the Unit of work pattern); or work directly with an entity (implementing the Repository pattern).

4.1.2. *Database*

Oracle Database 18c & SQL

Oracle Database is a database management system produced by the Oracle Corporation [73]. We use a relational database in which we store massive amounts of data that we need to access fast. Version 18c supports approximate query processing, which allows us to do approximate queries that respond in a shorter amount of time than typical, traditional queries.

Structured Query Language (SQL) is a standard language used for communicating with relational databases management systems. [74]

The Oracle Database was chosen primarily for the approximate query processing feature.

Microsoft SQL Server 2017 & LINQ

Microsoft SQL Server is a relational database management system developed by Microsoft [75]. We use a relational database to store all our business objects.

Language Integrated Query (LINQ) [76] is a Microsoft .Net Framework component that adds native data querying capabilities to .Net Languages. It is uniform query syntax used to retrieve data from different sources and formats, from lists of in-memory objects, to databases.

LINQ is used for communicating with the database server, but SQL commands are used behind the scenes. LINQ code and lambdas are automatically converted into SQL queries for the database by the data access framework we use.

Microsoft SQL Server was chosen for the good integration with .Net Core and EF Core frameworks, which simplifies development and maintenance.

Redis

Redis is a distributed, in-memory key-value store with optional durability [77]. It can be used as a database, cache or message broker. We are using the cache functionality for storing transient data, with enabled expiration on the objects stored.

I chose Redis because of the ease of set up and my familiarity with the technology.

4.1.3. *Frontend*

TypeScript

TypeScript is a typed superset of JavaScript that compiles into plain JavaScript [78]. That means all JavaScript code is considered TypeScript code as well.

Typescript helps with maintainability and makes sure the code is safe at runtime. It is type safe as the compiler checks at build time if the correct types are used in the project. In contrast to JavaScript, it offers the possibility of using objectual programming features, such as classes, making the code more readable and understandable.

Angular Framework

Angular is a TypeScript based open-source web application framework. We choose Angular because it's cross platform, the same code base works on web views and as native mobile applications. Also, it scales well for big applications [79]. We are using version 6.1 of Angular.

For downloading third party components and packages in the frontend we are using npm [80] and we are using webpack for building our angular application [81].

HTML 5 & CSS

We used HTML5 [82], as it's the current latest standard for web pages, and CSS [83] for styling the webpages. As a default style we used Bootswatch [84], which is a free Bootstrap [85] theme.

4.1.4. *Other*

Towards the end of the programming lifecycle we used Visual Studio 2019 Preview and the Visual Studio 2019 Community [66].

Other tools used in development:

- Visual Studio Code – is a source-code editor developed by Microsoft. It's very lightweight, compared to Visual Studio, and it offers dynamically plug in support for different languages and technologies used [86]. We sometimes used this IDE for developing the Angular code, as it's more lightweight than Visual Studio Community.
- SQL Server Management Studio – (SSMS) is a software application developed by Microsoft for configuring, managing and administering the components inside Microsoft SQL Server [87]. We used this tool for checking and altering data in the SQL database running on Microsoft SQL Server.
- SQL Developer – is an IDE developed by Oracle, used for working with SQL in Oracle Databases [88]. We used this tool for checking and altering data in the SQL database running on Oracle Database Server.
- Notepad ++ - is a lightweight source code editor [89]. We used this for visualising JSON and XML files.
- SqlDbm – is a web application used for database design [90]. We used this application for building our initial SQL diagrams.
- AWS – Amazon Web Service provides instances for running database and computational servers [91]. We used AWS for running production instances for the databases use and for testing the production code on a server.
- Draw.io – a web application that we used for creating figures in this thesis [92] as it's a free, online tool.

4.2. Data management

This section focuses on the implementation of the Data Management Service and other similar services used in the application. I present how the data is stored and accessed, and how all the systems are set up.

The Data Management Service uses persistent storage for handling user and nudge data, storing user information and history about previous nudges with trip and weather background context. For take advantage of APQ support we chose to store a copy of the Nudge Table in a different database for technical reasons. The initially chosen database server does not yet support AQP, so the nudge history data is copied in a database that does. The decision not to completely change the database provider was made because the database with AQP support does not directly support our current technologies, so tailored methods for the current needs had to be made. This is a temporary solution.

It is also using an in-memory data store for holding data a short amount of time, to improve response time when data from external providers is requested.

Data management code is running both on the server side and on the client side, on the client application. The weather forecast data is provided by AccuWeather [93], the bus schedules by Tromskortet [94], directions data by Google Directions API [95] and calendar events information by Google Calendar API [96].

4.2.1. *Setup*

This section explains how the different kinds of storage systems are set up. Mainly I present the two different types of persistent storage and at the In-Memory data store.

4.2.1.1. Persistent storage

The persistent storage is handled in two ways in the application, using two different SQL database providers. This design choice is made for technical and performance reasons. The SQL servers are running on our local machines in development mode, and on two different machines in production. The decision is made at compile time, checking if it's compiled in debug or in release mode.

The two SQL server systems used are:

- Microsoft SQL Server – All persistent data is stored in a relational database. This is the main storage system. The system is used because of the support offered by EF Core, the data access framework, making it easier to programmatically handle the data. EF Core offers objects and methods for simplifying database – code integrations.
- Oracle Database Server – A copy of the nudge data is stored in a database with a single table. This is chosen for performance reasons, this server supporting approximate query processing, being able to perform approximate query count on the table faster than normal queries.

The two SQL system approach is temporary, as new versions are being released in 2019 with Microsoft SQL Server support for AQP.

Microsoft SQL Server

The application needs a connection to an instance of Microsoft SQL Server to be able to run. The instance can run either on the same machine as the application or on a different one. Microsoft SQL Server 2017 is used, the latest released version of SQL Server during the implementation. When running the instance on the local machine the application uses Windows authentication for accessing the database, as it is developed under the Windows operating system. When connecting to an instance running on a separate machine, SQL authentication needs to be used.

The application uses EF Core to connect to the SQL Server. The framework creates a layer over the database, providing methods for connection and accessing the data. It offers two ways of manipulating the data: either using the database context, which has access to all tables in the database, offering the possibility of batching multiple transaction, or using a database entity, essentially accessing one table in the database.

The framework handles the actual creation and modification of the tables in the database. Each table has a representation in the code as an entity class, with each table column represented by class properties. Data access is performed using a model, which is formed by the entity classes and the database context, representing a session with the database. [72]

For representing relationships between tables, virtual properties of the entity class type are used to represent the actual foreign entries, and an Id property to represent the foreign key.

For creating or altering the database, *Migrations* are generated using shell commands. They are generated based on the model representation in the code. The migrations can be applied during application start-up or using shell commands. When triggered, EF Core connects to the database server and checks a special table in the database that holds all the applied migrations. If there are any unapplied ones, the framework alters the database on based on the new migrations.

EF Core is provided with a connection string that contains the details about where the database server resides and what credentials to use when connecting. When running in development mode, the application connects to the database server running on the local machine using Windows authentication. When running in production mode the connection string is set up to a different machine and includes SQL connection credentials.

Oracle Database Server

For the application to take advantage of all the features, a connection to an Oracle Database Server is required. This offers the application access to AQP techniques, for faster data access when faced with Big Data compared to traditional querying methods, as explained in section 2.5 of the Background. The SQL Server can run either on the local machine or on a separate one. For this implementation Oracle Database 18c is used with SQL authentication.

For facilitating data access, *Managed Data Access* NuGet package provided by Oracle is used [97], which offers some objects helping with database connection. The code implementation manually opens connections, applies transactions and queries the database server. The table in database is created and queried manually using SQL commands.

There are no relationships being maintained in this database, as there is only one table being used to hold a copy of the Nudge Table.

Similar to Microsoft SQL Server, 2 different methods are used for connecting to the Database Server. When running in development mode the connection is made to an instance of the server running on the local machine, while when running in production mode, the application connects to a separate one. A connection string is used, which holds the information about the location of the instance, being it on local or separate machine, and SQL authentication credentials.

4.2.1.2. Transient storage

An instance of Redis is required to be running on the same machine as the application. The application uses the In-Memory Key-value data store functionality of Redis.

The connection to Redis is done during the application start-up using a string with connection details. As Redis is used on the local machine with no special security, the connection to the instance is made with no authentication methods.

4.2.2. *Accessing stored data*

This section focuses on how the data stored in our systems is accessed. This represents both data stored persistently and data held temporary in memory.

4.2.2.1. Persistent data

For accessing data stored persistently the Repository coding pattern is used. This means using special classes to hide the database access methods from the application code. Using this pattern is helpful, as two different database servers with two different accessing modes are used, in simplifying the code migration to one or the other in future development.

Microsoft SQL Server Access

All the user and nudge data are stored persistently in a Microsoft SQL Server database. All the access methods are provided by EF Core.

For accessing and filtering the data LINQ commands are used, which offer a common fashion for generating SQL queries as for manipulating lists. From code perspective, filtering lists or database table entries is the same thing. The service requests an *IQueriable* object from the database context on which it applies different filtering methods using *Where* commands. A representation of the LINQ query is passed to the database provider to be translated into SQL [98]. Upon conversion to list, a list of all the resulted table entries is received. The database provider can be instructed to include the objects that have relationships with the queried entry, for example, when requesting the user, the account entry for that user may be necessary to be included for accessing the user stored credentials.

Inserting an entry in the database is done by creating an instance of a database entity and giving it to the database context. The actual insert is done when the database context is saved, offering the possibility of batching multiple inserts. Upon saving, the SQL code is generated and sent to the database server.

Updating existing entries is done by querying the required entries while instructing the database context to maintain tracking for them. This keeps track of any modifications done to the resulted entity objects. When a database context save is done, SQL update commands are generated to alter the modified entries.

For maintaining the two auditable columns, the Created on and Modified on columns, special methods were implemented that work to maintain correct values. When creating a new entity object, the entries are automatically provided with an Id and the current date and time for the Created on property, as it is considered that the time between the creation of the entity and the actual insert is insignificant. For maintaining the Modified on the Update method is overridden so that it sets the current date and time before saving the context.

A database context save can be done after multiple inserts and updates to the database context, batching multiple SQL commands together, to increase speed, as the framework doesn't need to wait for each command to finish to send the next one.

Oracle Database Server Access

For accessing the copied nudge information from the Oracle database, pure SQL commands are used, generated by implemented methods. The service performs only Insert and Select commands, using just APPROX_COUNT_DISTINCT to count the number of distinct users after applying filtering.

The same nudge entity object is used as for the main storage, in the Microsoft SQL Server. The Data Management Service programmatically creates the Insert SQL commands in plain text with all the values of the properties and send it to the database server. Each command sent one by one, with no batching applied.

For performing selects the APPROX_COUNT_DISTINCT command is used on the User id property. This returns an approximation of the result which should still be valuable for analysis when faced with Big Data, while having shorter response times than traditional count distinct commands, as explained in section 2.5.

For applying filtering on the select commands, a *QueryFilter* object was implemented that contains all the properties of the nudge table that can be filtered on. All values are nullable, skipping filtering for null values. When generating the query, all the properties of the *QueryFilter* type are considered, checking which properties of the object instance has values. For the ones that do, the property name is checked. If it starts with *Min* or *Max* the appropriate filtering condition is added. For property names with no keywords the equality filtering condition is added. The filtering conditions are inclusive, all filtering must be met on the resulted entries.

After the filtering is added to the select command, it is sent to the database server to execute.

Delete and Update methods are not necessary in this implementation. The delete methods may be necessary in the future to remove old, stale data.

4.2.2.2. Transient data

Transient data represents data that is temporary stored in memory for a short amount of time, especially for performance reasons. Availability for the stored data is not ensured, as it can be deleted from memory when its lifetime expires. The data is stored in the Redis In-Memory Key-Value store.

For accessing and manipulating this data a helper service named the Memory Cache Service was implemented. This service handles all issues relating to storing and retrieving data from the data store.

Inserting an object in the data store requires a key that is easy to generate, as the service does not offer access to stored keys for later retrieval. The Redis server is capable of storing byte arrays, so the Memory Cache Service converts the object to be stored into a byte array before insertion. Upon insert, beside the key and the object to be stored, the service can optionally be provided with the object's lifespan, which indicates after what amount of time the object is deleted from the store. If no such lifespan is provided, the default is set to one hour. The Redis server handles automatic deletion upon lifespan expiration.

Retrieving the object is done by providing the service with the key to be retrieved and the object type, for conversion from byte array. The Memory Cache Service handles the conversion to the required object type before providing the result.

4.2.3. *Accessing external data*

This section focuses on how data is retrieved from external providers. The Data Management Service accesses user location, weather forecast, current weather conditions data, calendar event information, bus schedules and travel directions information.

For facilitating requests over the web, *RestSharp* NuGet package was used, which offers methods for generating http requests.

The user's location is provided by the browser, with the user's permission.

4.2.3.1. Weather forecast

AccuWeather is used as the weather forecast provider. The free plan was opted for, which provides a set of APIs, for requesting the current conditions, a 12-hour forecast and a list of locations where the forecast service is provided. The free plan limits requests at 50 per day, which should be enough for the current needs.

Two types of requests are used: the current conditions for when a user needs to travel now, and the 12-hour forecast for when a user needs to travel for an appointment in the future.

For retrieving the current conditions AccuWeather provides detailed current forecast information and some historical data as JSON objects. The Data Management Service makes an http request to the current conditions API, retrieving the JSON objects and converting them into application objects. Light filtering and conversions are applied on the received data to remove what is unwanted and merge certain information into more readable forms, instead of keeping the wind speed, the data is converted into a representation for *strong winds*, for example.

For retrieving the next 12-hour forecast, a similar http request is made to the appropriate API retrieving some JSON objects with forecast information for each hour. The objects are converted into application specific objects and filtering is applied for getting the required hour for which the forecast is needed. Before providing the result, light processing and filtering is applied in a similar manner as for the current conditions.

4.2.3.2. Bus schedules

For getting the bus schedule information in the Troms area the Tromskortet service is used, which provides free RESTful APIs for querying the nearest bus stops and the bus schedules.

When a user travels with the bus there is some information required, that is: the nearest bus stop to the user, what bus he needs to take, when is the bus arriving, if there are any connections, where is the nearest bus stop to his destination and when is the bus arriving there.

For retrieving the nearest stops for both the user's location and the destination, the Data Management Service makes an http request to the nearest stops API, providing the coordinates of either the user or the destination. The API responds with a list of XML objects representing the bus stops ordered from nearest to furthest. The received objects are converted into application objects.

For retrieving the bus schedules between two stops the service makes an http request to the provided Search API, with the departure and arrival bus stop names attached, and either the date and time when the bus departs or when it arrives at the destination. The API responds with a list of XML objects representing a list of possible bus routes, with bus or bus stop changes if necessary. The service converts these objects into application objects and returns the first provided route.

4.2.3.3. Google APIs

For retrieving calendar, locations, and walking and cycling route information, APIs provided by Google are used. These services are facilitated using HTTP requests.

The Data Management Service uses the public Google calendar service to access the Norwegian holiday list. This service is requested in the backend by a daily running task. The holiday events are converted upon receival into objects containing only the event summary, start and end date.

Accessing the user's calendar is done in the frontend, upon user's authentication with Google, with the user's permission. The events are filtered to handle only the ones that have an event location, so the Data Analysis Service is able to nudge with travel information to the address of the event.

Requests for cycling and walking directions for the user trip is done in the frontend using the *Distance Matrix Service* provided by Google, while directions for walking from the user's location to the nearest bus stop or from the bus stop to the destination is done in the backend by the Data Management Service using the same Google service.

The service also uses the Locations API to retrieve the coordinates for the specific address. This API is needed when computing the bus travel route.

4.2.4. *Storing user information*

The Data Management Service handles the storage and access of user information. This represents user preferences, address, email and credentials.

There are two ways for user authentication: local and google login. The user address and preferences are stored when authenticating with local login, on the other hand, when authenticating with Google, the registration page is skipped, and the service saves only the email and marks the Google authentication in persistent storage.

Once authenticated, the client app asks permission for sending notifications and sends the notification endpoint information to the data management service for storage.

Also, the user can provide the application information about his preferences, such as the temperature he doesn't mind being nudged to go walking or cycling, if it can be rainy or snowy, and his preferred mode of transport.

4.2.5. *Storing nudges*

Nudges are stored for future analysis to provide a historical view for how the users acted upon previous nudges. The nudge context represents the trip information and the weather forecast. Trip information includes what mode of transport it was nudged upon and the trip duration and distance. Weather forecast represents the temperature, the precipitation, cloud coverage and conditions for that day.

When the data analysis service decides to send a nudge, the nudge is stored, along with the nudge context, in an unknown result state. This kind of nudges are not useful for analysis, as no information is provided yet about how the user reacted to the nudge.

Later, the user can offer input on the nudge notification in a special page, either the notification history page or the nudge details page, the nudge result state being updated to *Successful* or *Failed*.

The trip data is optional, as there are nudges that do not provide trip information, they nudge just for the user to go for a walk, if it's a weekend or a public holiday. In such cases, the type of the nudge is important, as it is a special type. This information is interesting to know, if the users responds well or not to this kind of nudges.

4.2.6. *Providing trip information*

For providing trip information, user location, destination and time of departure or arrival are necessary. The Data Management Service provides routes for walking, cycling or traveling by bus.

The user coordinates are provided by the browser and the destination is provided by the user in plain text through the Client App.

4.2.6.1. Walking & Cycling

Walking and cycling is handled on the client side of the application, on the Client App, using Google Directions API. The API is provided with the user's location, his destination and the travel mode. In turn, it responds with a list of results with different travel routes. As the default first provided result is presented.

Similar APIs are implemented on the server side for the Data Analysis Service. They are implemented under the Data Management Service and they use *RestSharp* for making request to the Google APIs.

4.2.6.2. Traveling by bus

Getting bus route information is handled in the backend by the Data Management Service. Usually, a bus trip has steps similar to the ones presented in Figure 9. The user at location **A** needs to walk to the bus stop **B**, where he takes the bus until bus stop **C** from which he needs to walk to his destination **D**. The bus trip **B – C** can be formed of multiple session of walking, if the user needs to change the bus stop once during the trip, or simply waiting for a connection. The Data Management Service takes all of this into account when computing the bus trip.

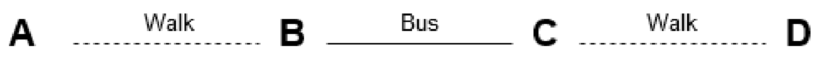


Figure 9 Bus trip

There are two modes for providing the data, when the user leaves now or when he has an appointment. Both approaches are similar, with a few differences in computing the result.

The service gets the nearest stops to the user and to his destination and requests a bus route between the two stops. Based on this data a trip object is built with multiple trip parts, the first part considered the walk from the user's location to the bus stop, the second part representing the bus trip to the destination stop or the stop where the user needs to change the bus, and so on. The last trip part represents the walk from the last bus stop to the destination. The resulted object contains all the trip parts, the total duration, the departure time. the arrival time and a link with to the Tromskortet website for that trip.

If the user intends of leaving now, the departure time will be set to the time he can take the first bus, giving him enough time to walk to the bus stop. Walk duration information is provided through Google Directions API.

If the user has an appointment, the departure time is calculated by requesting the time the latest bus arrives at the bus destination, giving the user enough time to walk from the bus stop to the destination.

Obviously, this approach has its limitations, as the service takes into account only the closest bus stops to the user's location and destination. Better routes may be provided if the service checks multiple stops.

4.2.7. *Providing weather information*

Weather information is provided for Tromsø only, both for the weather current conditions and a 12-hour forecast.

The Data Management Service checks with the Memory Cache Service if there is data already stored in memory using a key generated with the current hour. If a key-value pair exists, the data is served directly from memory. If the value is missing, a normal HTTP request is made, and the result saved in memory before serving the requester.

4.3. Application

This section focuses on the application as a whole and how the services work to handle the data: when, how and why the data is stored.

All application services setup is done during start-up. The application uses Dependency Injection for service interaction. On start-up the database context is registered, instructing it what SQL server to use, followed by all implemented services. During a service initialization, its constructor receives all dependent services.

During the start-up phase, the connection to the Redis In-Memory data store which resides on the local machine is set up.

After the setup is finished, the .Net Core framework starts the frontend application. The application uses npm to download all required dependencies and webpack to trigger build events. After the dependencies are downloaded and the typescript compiler finishes converting Typescript files to JavaScript, the Angular framework takes control of the frontend application.

The frontend code is organised as follows:

- Components – one for each page displayed, handles how to show the information on the page and how to organise it.
- Services – handles getting data from the backend or from other data sources, helpers, etc.
- Types – contains our defined data types.

At angular start-up, it is defined which links are handled by which components. Also, here the API keys for google maps, calendar and directions APIs are prepared. An HTTP Interceptor is started during this phase, which intercepts the requests towards the backend server and adds the authorization token to them, if one is present in the browser cookies.

For simplicity and ease of use, most of the services are moved to the backend, on the application server. That means that the server will have knowledge about the user’s location and destination. Some of the services could be moved to the fronted, on the client application, to address privacy concerns.

4.3.1. *Authorization and Authentication*

Two different types of authentication are used in the application: local and google authentication. Local authentication offers limited functionality, as no calendar information is provided

Local authentication

The first iteration of the application was made with local logins, meaning that the application handles the authentication itself, storing the authentication credentials, while offering limited functionality.

During user registration, the application receives the password and generate a random password salt, representing a random string. Using the user’s password and the random salt, the password hash is generated, which is then stored in the Accounts table, along with the salt.

During the login phase, the application receives the password, hashes it using the stored password salt and verifies if the result and the stored hash are identical. If they are, we create an authentication token and send it back to the client app to attach it to all its http requests for the server for authorization. The client app stores this token using cookies in the browser.

Google authentication

For activating google authentication the application was registered with the Google dashboard, generating Google application credentials, with calendar access permissions.

When logging in, the client application makes a login request to Google, which handles te authentication and the calendar permission requests. If successful, Google sends to the application user information and an access token. To verify the authentication with our server, it verifies the access token with Google. If the verification is successful the Google authentication bit in the Accounts table is set, to mark that the user connected at least once with Google. Afterwards, the server sends to the client application the authentication token to store in the cookies.

4.3.2. *Data Analysis Service*

The Data Analysis Service runs based on scheduled events. It can be triggered either by a daily running task or by the frontend, when the user has an appointment.

This service asks for user history, weather data and looks at patterns for users in similar weather conditions to decide if a certain user should be nudged or not. If the service decides upon nudging, it uses the push notification service to send a notification to that particular user.

More data about the Data Analysis Service can be found in the Data Analysis and Nudging for Green Transportation by Jemea Lady Limunga. [1]

4.3.3. *Notification Service*

The Notification Service handles sending the nudge push notifications to the user. It is issued by the Data Analysis Service to send nudge notifications to the user.

Once the user logs into the application, it requests permissions to accept notifications and saves the endpoint where the notifications will be sent to, in addition to some required subscription authentication credentials, data stored in using the Data Management Service.

The push notification service, upon request, uses *WebPush* [70] to send a notification to an user. The notification text is provided, usually by the Data Analysis Service. The notification is sent to all the saved user endpoints, along with the endpoint authentication credentials.

The endpoint automatically checks the credentials and displays the notification to the user, title with message. When the notification service sends the notification, it also sends it to the Data Management Service for storage, as the user may offer feedback based on the notification.

4.3.4. *Application triggers*

The application itself has multiple triggers, both for storing and analysis. Either a user triggers it from the client application, or a different process running on the server or on the client application side triggers the analysis. We separate these into user triggered, scheduled and user calendar events.

User Triggered

The application can be used directly by the user to check destinations, options of traveling and checking forecast data. (Figure 10)

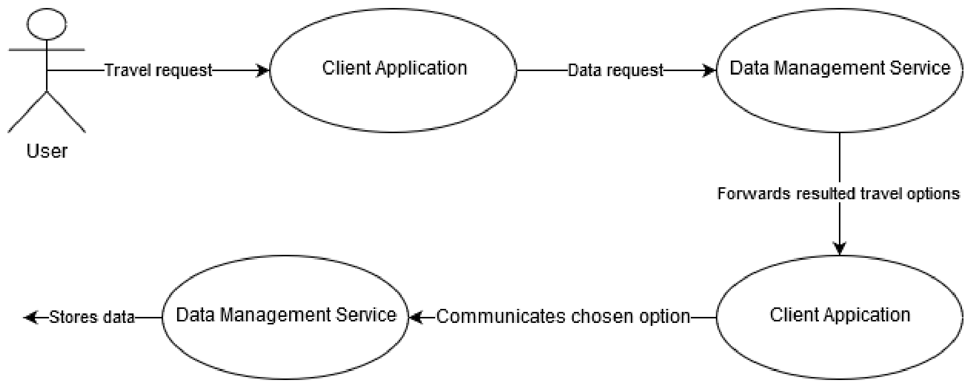


Figure 10 User Triggered Application Flow

If the user wants to commute to a location, he uses the client application to enter the address of his destination and triggers the *Travel Now* mode of the application. In this case, the application gets the user location from the device and gets the travel options for walking and cycling to his destination from the Google Maps service.

It also sends the user location and his destination to the data management service running on our server, to compute the bus travel data. The server gets the data from the providers, it filters and combines it and then returns the result to the user. The location data is used to retrieve forecast data as well, for presenting the current weather conditions.

The user is presented with 4 different options: walking, cycling, taking the bus or driving. We give details about the trip duration for the first three and the current temperature.

Once the user chooses one of the first three modes of transport, the option is sent to the data management service, which saves it as an accepted nudge. If the choice is driving, it is stored as a failed nudge. As for the user, he is redirected to the appropriate external page, either the google maps or to the page with the bus schedule.

Scheduled Events

The application also runs based on scheduled events. Each task is set up with a schedule for running, either at specific times, or once every certain amount of time. The implemented task scheduler runs continuously in the background, checking if the assigned task should run or not, depending on the current time. *NCronTab* NuGet package [99] is used to get the task schedule. The tasks are registered with dependency injection as an *IHostedService*, which are run automatically by the .Net Framework when the application starts.

The *SpareTimeNudgeTask* was implemented as a hosted service, which runs only in the weekends, at 10 in the morning. It checks the weather forecast in Tromsø and, depending on the weather conditions, nudges the users to go for a walk outside, in the park for example.

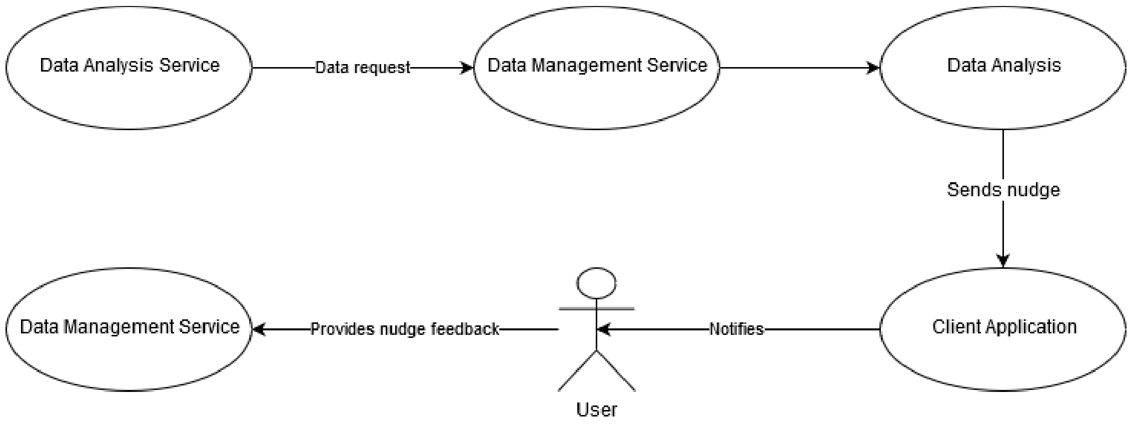


Figure 11 Event triggered Application Flow

The Data Analysis Service is triggered by the event and requests data from the Data Management Service. Depending on the received data, the Data Analysis may decide to send a nudge to the Client Application to notify the user. The user may later choose to provide feedback on the nudge notification by setting the nudge result, successful or not, which is stored by the Data Management Service. The flow is represented in Figure 11.

User Calendar Events

The third way the application is triggered is when the client application reads a user calendar event with a location address. In this case, the client application informs the Data Analysis Service about the time and location of the event. The service requests more data from the Data Management Service about user history, environmental information, etc., and then triggers a nudge, if necessary.

Essentially, this trigger is similar to the Scheduled Events trigger, as the flow is very similar. The only difference is that the Data Analysis Service is triggered from outside and knows time and location information about the event and the user's location.

4.3.5. *Nudge feedback*

A nudge notification is triggered by an event, either by a public holiday event or by a user event. When the Data Analysis Service decides to notify, the nudge context is saved, and a push notification is sent to the user.

Later, the user can look in the Application, in a special page, to see all former nudge notifications and set their state, either *Successful* or *Failed*. He can also have a more detailed view of the nudge, displaying the saved context information, as the trip duration and length, and as the weather conditions for the trip.

4.4. Considerations

4.4.1. *SQL Servers*

Two SQL servers were used in this implementation as a temporary solution. Microsoft SQL Server was used because of the great support offered by EF Core. Version 2017 did not offer support for AQP, for that reason, it was chosen to include an Oracle Database server with such support and keep a copy of the table where approximate queries are applied.

I say it is a temporary solution, as Microsoft SQL Server 2019 is in the Preview state in the time of the implementation. The newer version of the server comes with AQP support, which guarantees up to 2% error rate within 97% probability [100], in contrast of Oracle which is 95% confident that the approximate results are within + or - 3% of the answer [101].

4.4.2. *Troms area dependent*

The current implementation of the application is dependent on the Troms area as I am using a local bus service for receiving bus schedule information. Also, the weather data is hard coded to request data only for Tromsø.

The reason it was chosen to use the local bus service is that Google Maps has no information about bus schedules in Tromsø, like other places in Europe, at least at the time of the implementation.

In the AccuWeather service, each location is attributed a code, so for each location the data management service should get the location code before making the forecast request. This step was skipped by hardcoding the location code directly in the forecast request.

4.4.3. *User nudge feedback*

Notifications appear to the user without any possibility for him to interact with them, they just display the nudge text. For the user to provide feedback on the nudge he needs to go to the application page and browse to the nudge history page or to a particular nudge and set the nudge result from there.

This is a tedious process that most users probably won’t go though. He should be able to set the state from the notification or jump from it directly to the nudge details page.

We choose to go this method for the users to provide feedback on the nudge as a proof of concept for the application.

4.4.4. *Single Page Application*

We choose going for building a SPA with Angular as it can be modified to run on a native mobile application. Due to time constraints we didn’t develop it as one, rendering the final application just at proof of concept state. The notification system in the SPA works only as long as the user has the application running in the browser, opposite to a mobile version, which could run in the background.

For this reason, the user event triggered flow would not work as expected, as it runs only at log in for now. It should be a different process that runs in the background of the Client Application, checking the user’s calendar and triggering analysis.

5. Experimentation

In this section I analyse some of the design choices we made and compare the results with a more traditional approach. I look at how approximate query processing handles the data, compared to traditional querying, and I compare running times of forecasts using the in-memory store and without.

5.1. Approximate Query Comparison

In my setup, I am running an Oracle Database server on my local machine and I populate the database with nudge data that contains randomised values. By holding the database on the local machine, I avoid latencies generated from web communications. The ratio of the entries count to the distinct users is 0.1% (at 10 million entries, we have 1 000 distinct users; at 2 million entries, we have 200 distinct users). This ratio was chosen as it's similar to the one given as an example [102].

Each sampling consists in 100 randomized queries, for which we compute the average execution time.

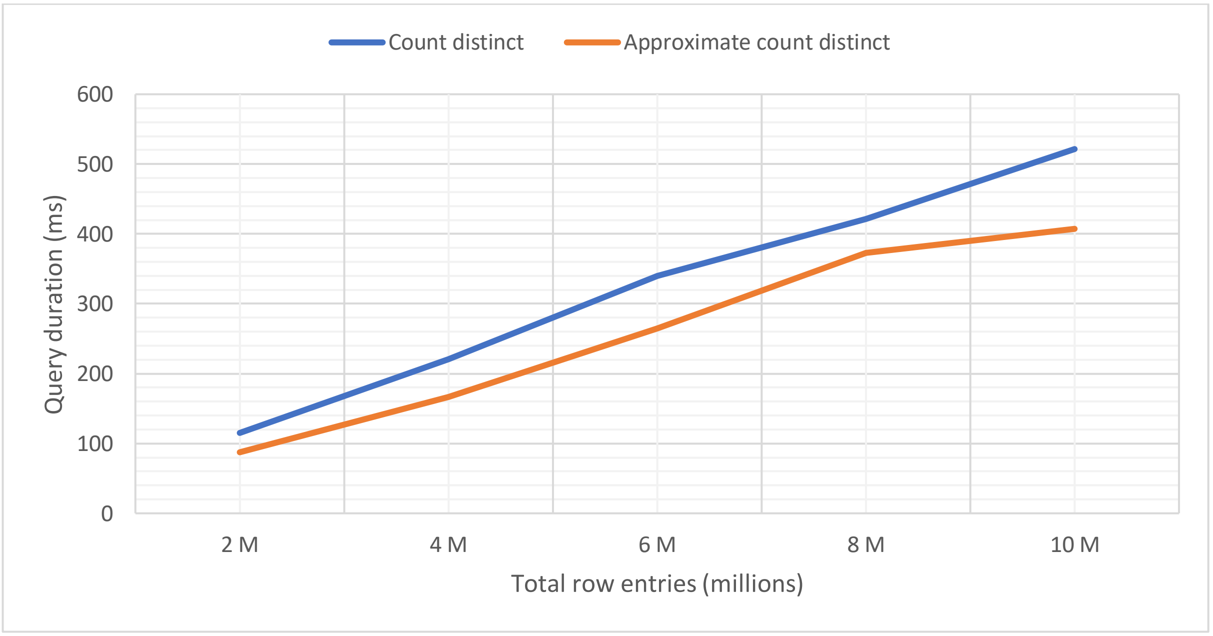


Figure 12 Count distinct vs Approximate count distinct (2 - 10 million entries)

As shown in figure 12 the approximate distinct count command has an increase in execution time but has generally a faster response rate. The effect is even stronger when going to higher values, the biggest effect is for 10 million entries and 1 thousand distinct users.

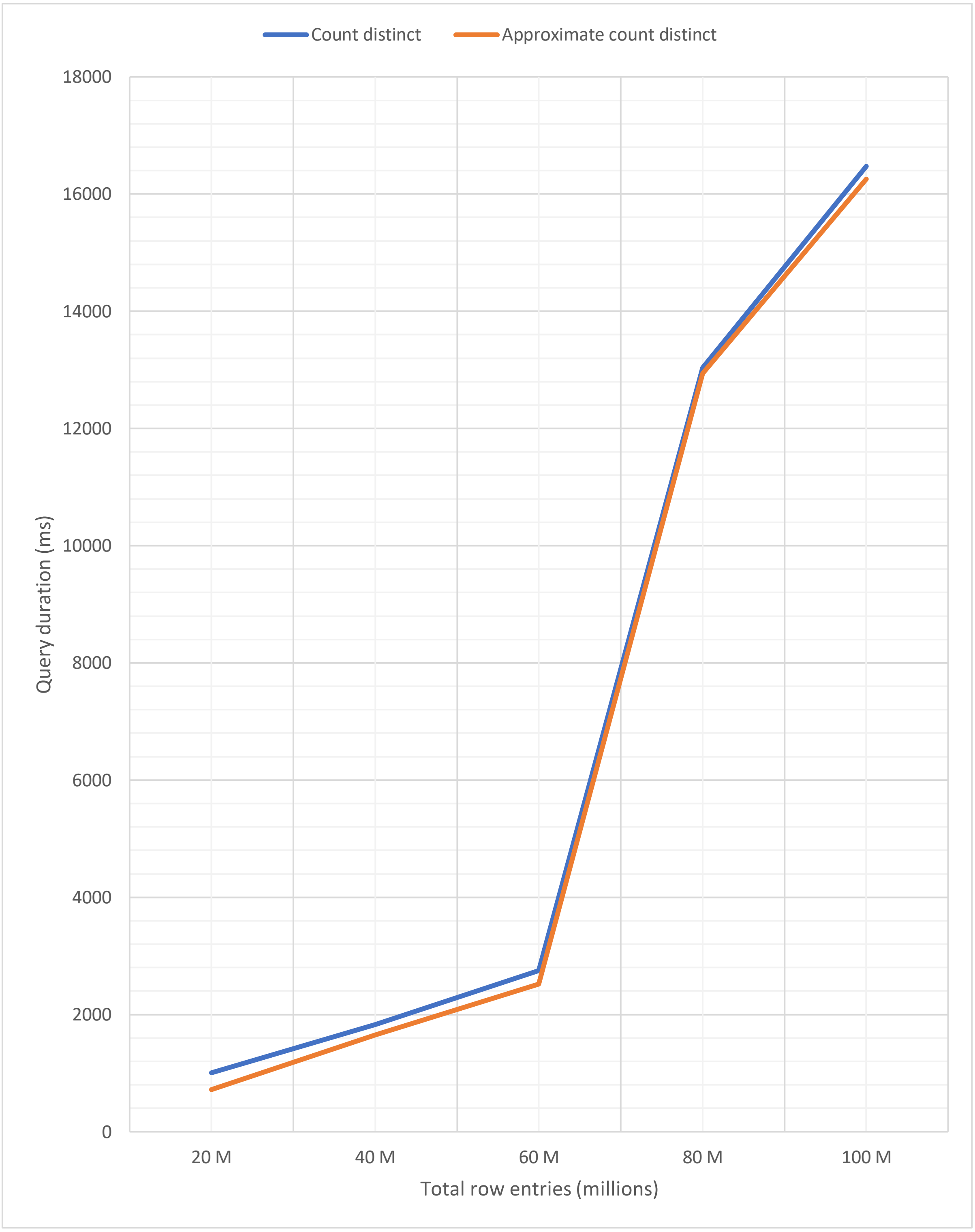


Figure 13 Count distinct vs Approximate count distinct (20 - 100 million entries)

When generating even bigger data sets, the performance difference is not seen anymore. In Figure 13, the approximate count distinct command is slightly faster than the traditional command, but as the data sets grow, the advantages are not evident anymore.

5.2. In-memory store

I used the in-memory data store as an improvement for reading transient data from external providers, specifically, when reading the current conditions from the weather provider. If the last request was made in less than one hour, the Data Management Service fetches the data from the memory, instead of requesting it over the web.

In the test setup we used the implementation of the weather forecast service is tested with and without the in-memory store. We measured the duration of the 50 requests and averaged them out.

Table 9 In-memory data store test results

	MEMORY HIT	**NO MEMORY**	**NORMAL USE**
AVERAGE RESPONSE TIME (MS)	1.06	374	383
REQUESTS PER SECOND	943.40	2.68	2.61

The results are shown in table 9. The memory hit column shows the response time when the forecast has been gotten directly from memory; the no memory column shows what a normal request would look like, with no in-memory store usage. The normal use represents our implementation, when we check in memory failing to find entry, make a normal REST request, and saving the result in memory.

As expected, the approach using the memory store is a bit slower in case of memory miss, but not enough to be an impact. The trade of is that, for future requests, with hit in memory, the response is nearly instantaneous.

6. Discussion

In this section we will focus on discussing the experimental results and our experience based on the design choices.

6.1. Oracle database and AQP

My presumption that approximate processing will prove helpful in accessing the data has proven true in some extent. The query results do not lose their validity, even if the results are not exact, but the response is not significantly faster.

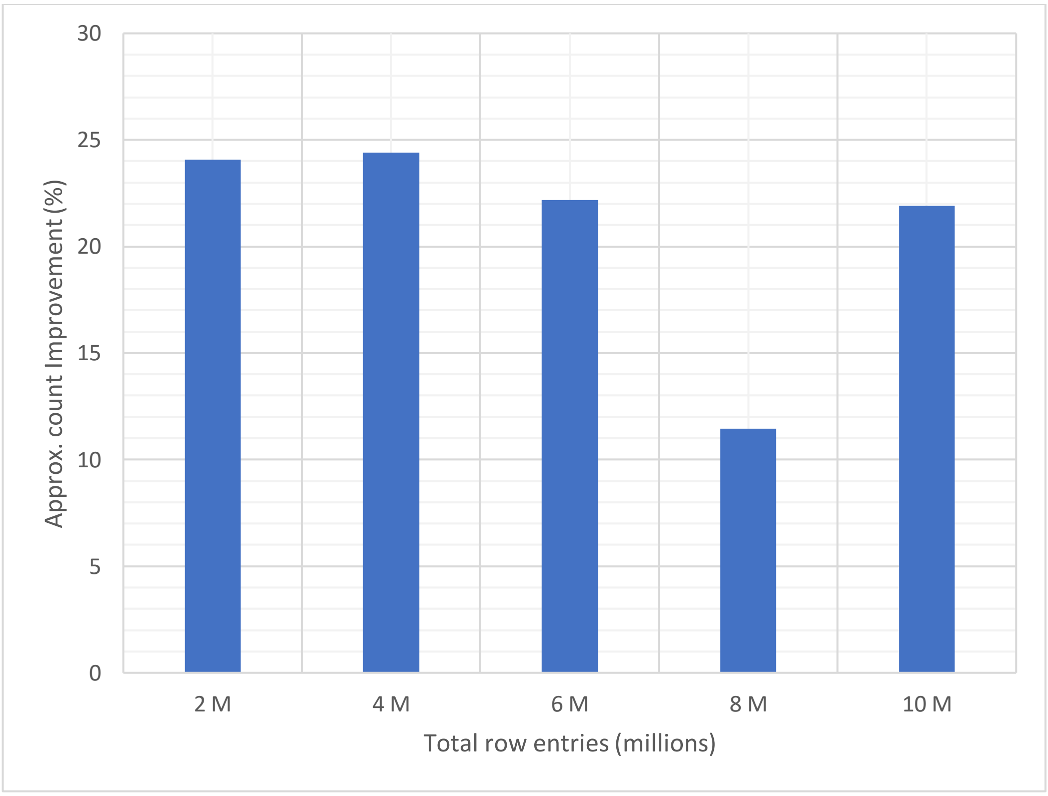


Figure 14 Approximate count distinct improvement (2 – 10 million entries)

Overall, for data sets ranging between 2 and 10 million entries, the improvement varies between 10% and 25%. As seen in Figure 14, each bar represents the improvement percentage over traditional count distinct command when the database has millions of entries. Each measurement was done after inserting 2 million more entries, 100 distinct users per million entries. The variation in improvement amount can be put upon the variety of queries, as each query is randomized. The average improvement is of 20.8%.

Compared to the tests run in [102], for lower data sets they state smaller differences 2.39 seconds for the traditional query and 2 seconds for the approximate one, which is a 19.5% improvement, similar to my results, but they do not state exactly the dimensions of data sets they used.

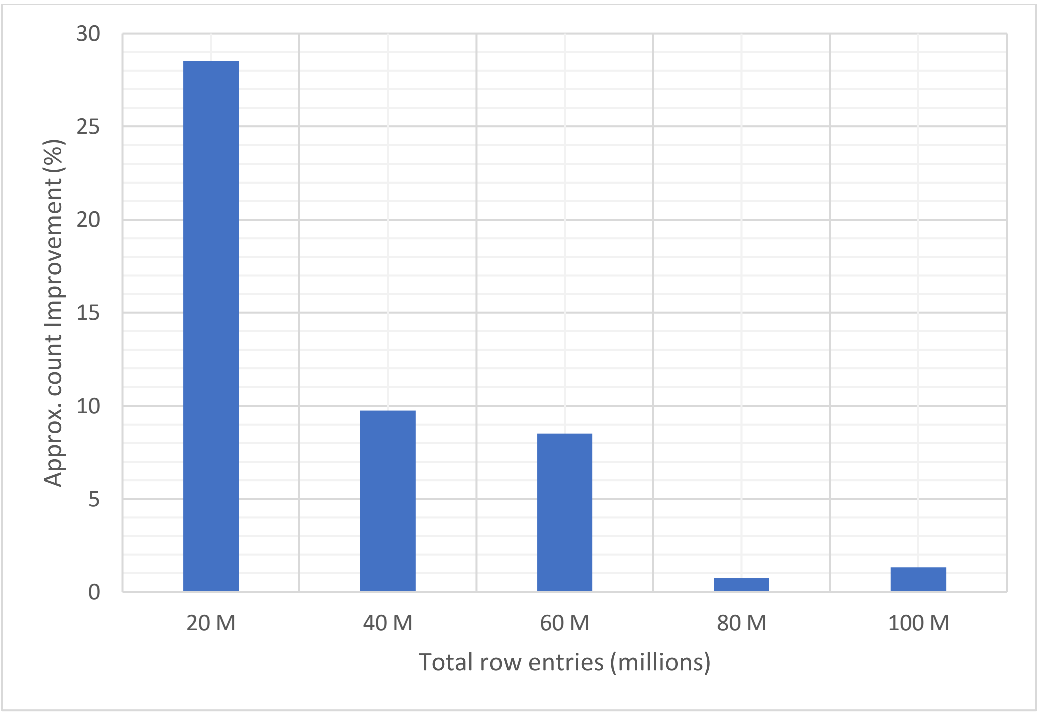


Figure 15 Approximate count distinct improvement (20 – 100 million entries)

In a similar experiment as the previous one, but with bigger data sets (Figure 15), the advantage of approximate queries is lost as data sets grow. The average improvement is of 9.75%, with the highest one for 20 million entries, of 28.5%. If not considering that, the average for 40 – 100 million entries is of 5%, with the lowest for 80 million entries at 0.7% and 100 million at 1.3%.

As opposed to [102], where they state the response speed for 100 million entries and 10 thousand distinct values as 19.6 seconds for a traditional query and 10 seconds for the approximate query, an improvement of almost 50%.

As seen in the results, the performance was usually over 20%, until the data set passed the 20 million count. This huge inconsistency after that can be explained because of the randomization of the data. The database used in [102] was Oracle Database 12c as opposed to 18c used by me. Also, the database was running on my local machine, which may have limited its performance, as opposed when running on a dedicated server.

I consider that using approximate processing offers some improvements and should be used in such an application as ours. Oracle states that they are 95% confident that the approximate results are within + or – 3% of the answer [101]. The table replication done for having this feature is not necessary in future versions, as the next version of Microsoft SQL Server supports approximate query processing out of the box [103] with even better guarantees: up to 2% error rate within 97% probability [100]. By moving completely to Microsoft SQL Server, we will have the benefits of relational and SQL databases with approximate processing features.

6.2. In-memory data store

In my implementation of the Data Management Service using the data store, when the application makes a request for the weather, the response is stored for an hour in the data store, with the current date and time as the key. The process of checking for the data, making the request and then saving is about 2.5% slower than a request without using the memory. The benefit is that later, forecast requests in the 1-hour timeframe are served from memory, significantly improving response times.

This implementation currently works only for Tromsø, where all users are expected to be. For a more general implementation, the key should be generated adding the location as well, having location and time in the key.

This is fine, as long as we can fairly guarantee that the data doesn't change at the provider. For the current conditions, storing for one hour is too much, as the conditions may rapidly change. But for other more static data we may be able to store and serve from memory.

6.3. Other design choices

Data Management and Data Analysis service

We chose to separate the data management service from the data analysis service because they serve different purposes. The Data Management Service handles storing and accessing the data, while Data Analysis oversees the analysis and decision making based on the results of the analysis.

Having two separate services offers better maintainability, as the code regarding one aspect of the application (store, access) is separate from the analysis. The Data Analysis Service is defined as requiring high processing power for analysing huge data sets, while the Data Management Service needs to focus on offering these data sets.

Also, both data analysis and management are expected to grow in the future. The Data Management Service will have to handle more data sources and do it more efficiently, while the Data Analysis Service's logic will grow to be able to handle the amount of data to be analysed. This brings scaling to mind. For now, both services are running on the same process, but they can be separated as 2 different applications, on 2 different processes, either using REST communication or a message broker system.

Nudge responses

A debate in our team during application development has been how do we know if the nudge notification has a positive effect on the user or not.

One method, based on a similar application (UbiGreen [63]), is to nudge and then track and see how the user is traveling based on sensor readings. We chose a simpler method, trusting the user to tell us if he accepts or rejects the nudge. We argue that this way we maintain a stronger level of privacy and as we assume that the user uses this application to help him change his normal transportation routines, it's not in his benefit to lie.

User preferences

As an initial form of our application, we expect the user to set his transportation preferences upon registration, but these can change over time, and the user may not update this. I am arguing the fact that the user preferences can be generated based on user activity. We should be able to tell the user's preferred mode of travel without explicit help from the user, either by tracking the activity or by the accepted or rejected nudge history.

Privacy

We didn't have privacy in mind while implementing this application, but I do have some thoughts on how to approach it.

Our current approach is to implement the services in the backend, with them running on the server, while the client app sends the user location, destination and other user information to our servers to compute nudges. To address privacy concerns, most of these services can be moved to the client side, if the necessary computation power is not high.

Services as the weather service and the bus service can be handled on the client side as well, as in certain cases they only present data to the user. When they are needed for analysis, we can have the option in triggering it from the client side, providing our server with the data it needs, like trip duration, weather conditions, etc., while avoiding specific user locations or destinations. Data analysis may need more computational power, as it sometimes needs to take in consideration other users and user historical data, so I suggest keeping it on the server side.

Relational Database

I chose to keep all persistent data in a relational database instead of opting for a NoSQL solution. I chose this to take advantage of the benefits of the relational model, which makes data handling easier.

I suggested in the background that a data warehouse is useful for storing data destined for analysis, but in our case that is not the case, as recent data is more useful than old one. Recent user activity is more relevant than old one, users change their behaviour over time.

7. Future Work

In this section I describe what further steps are needed for the application.

7.1. Updating Server version

Replicating the Nudge table is not optimal and was decided as a temporary approach. Upon Microsoft SQL Server 2019 release approximate processing will be available [103]. Using 2 database providers is no longer necessary and using only Microsoft SQL Server will be a huge improvement, providing the power of Entity Framework Core with approximate query features.

7.2. Changing the Relational Database

This point excludes the former one, as updating Microsoft SQL Server will no longer be necessary.

Oracle will soon support Entity Framework Core [104], which gives us the opportunity to choose between 2 database providers, both providing the relational system and approximate query capabilities.

Choosing Oracle going onward, we can move the whole database into Oracle Database, dropping Microsoft SQL Server entirely.

7.3. Adding more data sets

Several other data sets are needed for a better user experience and for better nudging parameters. More traffic, parking and road maintenance data is necessary, to update the user of possible problems in taking his personal car.

For the Tromsø region, snow track preparation is an interesting parameter, as people use skis to get around. People would be interested in taking the skis to their destination instead of any other mode of transport.

7.4. Mobile application

More extensive work needs to be done for a true mobile application, as the main work was done as a single page web application. We used the angular framework which helps moving the web application in the mobile environment.

8. Conclusion

Our approach for solving the problem of high greenhouse gas emissions generated by the transportation sector has resulted in the Nudge App, a prototype application that uses nudge theory for helping people choose greener methods of transport. Storing and accessing the data for analysis is a challenge, in which approximate query processing and caching can help. Also, nudging requires data from multiple sources to be able to build the nudging context for the user to achieve a successful nudge, this implies that the data required is massive and heterogeneous.

Applying light processing to filter what is not needed on the received data before storing resulted in success. The application handles data from multiple sources: user location, weather data, travel directions, bus schedules and calendar information. Based on this data, nudges are sent to users and stored in a nudge table containing the nudge result, trip and weather forecast information relevant to the nudge. This table is used to create a user nudge history for deciding on future nudges.

Approximate query methods were used on the nudge history table, resulting in better counting results over the huge data sets but offering less benefits than expected. The benefits of AQP were constant despite the data set growth, until it reached a point when the performance almost disappeared either due to limitations of the local machine or to the randomised data sets.

Caching was implemented using the Redis In-memory data store, with tremendous benefits. In case of hit in memory, the data response is nearly instantaneous, while in case of miss the response time is slightly slower, almost insignificant, than a normal request with no caching. The caching methods were used in the weather forecast data, caching the results for a certain time and serving the data from memory. It is important to be careful with such caching, as the data can rapidly become stale, losing its value.

My main contribution with this thesis is an example application on how to manage nudges for green transportation, with the focus on data management.

9. References

1. Limunga, J.L., *Data Analysis and Nudging for Green Transportation*. 2019, University of Tromsø.
2. Ganopolski, A., *Anthropogenic global warming: evidence, predictions and consequences*, in *Advances in Solar Energy: Volume 16*. 2017, Routledge. p. 1-34.
3. Hansen, J., et al., *Global surface temperature change.* Reviews of Geophysics, 2010. **48**(4).
4. Herring, S.C., et al., *Introduction to Explaining Extreme Events of 2017 from a Climate Perspective.* Bulletin of the American Meteorological Society, 2019. **100**(1): p. S1-S4.
5. Donnelly, J.P. and J.D. Woodruff, *Intense hurricane activity over the past 5,000 years controlled by El Niño and the West African monsoon.* Nature, 2007. **447**(7143): p. 465.
6. Sharp, M., et al., *Extreme melt on Canada's Arctic ice caps in the 21st century.* Geophysical Research Letters, 2011. **38**(11).
7. Meehl, G.A., et al., *How much more global warming and sea level rise?* science, 2005. **307**(5716): p. 1769-1772.
8. McGranahan, G., D. Balk, and B. Anderson, *The rising tide: assessing the risks of climate change and human settlements in low elevation coastal zones.* Environment and urbanization, 2007. **19**(1): p. 17-37.
9. Holmberg, K. and A. Erdemir, *Influence of tribology on global energy consumption, costs and emissions.* Friction, 2017. **5**(3): p. 263-284.
10. Herzog, H. and D. Golomb, *Carbon capture and storage from fossil fuel use.* Encyclopedia of energy, 2004. **1**(6562): p. 277-287.
11. van Alphen, K., M.P. Hekkert, and W.C. Turkenburg, *Comparing the development and deployment of carbon capture and storage technologies in Norway, the Netherlands, Australia, Canada and the United States–An innovation system perspective.* Energy Procedia, 2009. **1**(1): p. 4591-4599.
12. Sheehan, J., et al., *An overview of biodiesel and petroleum diesel life cycles*. 2000, National Renewable Energy Lab., Golden, CO (US).
13. Laustsen, J., *Energy Efficiency Requirements in Building Codes, Energy Efficiency Policies for New Buildings: IEA Information Paper.* Support of the G8 Plan of Action, 2008.
14. Ajanovic, A. and R. Haas, *Dissemination of electric vehicles in urban areas: Major factors for success.* Energy, 2016. **115**: p. 1451-1458.
15. MacKenzie, J.J. and M.P. Walsh, *Driving forces: motor vehicle trends and their implications for global warming energy strategies and transportation planning.* 1990.
16. Andersen, A., R. Karlsen, and W. Yu, *Green Transportation Choices with IoT and Smart Nudging*, in *Handbook of Smart Cities*. 2018, Springer. p. 331-354.
17. Wash, R., L. Hemphill, and P. Resnick. *Design decisions in the RideNow project*. in *Proceedings of the 2005 international ACM SIGGROUP conference on Supporting group work*. 2005. ACM.
18. Cirianni, F.M. and G. Leonardi, *Analysis of transport modes in the urban environment: An application for a sustainable mobility system.* WIT Transactions on Ecology and the Environment, 2006. **93**.
19. Basbas, S., *Sustainable urban mobility: the role of bus priority measures.* WIT Transactions on Ecology and the Environment, 2007. **102**.
20. Cityhall, B. *Protocol for air pollution episodes*. 2017 [cited 2019 May 8]; Available from: https://ajuntament.barcelona.cat/qualitataire/en/noticia/the-protocol-for-air-pollution-episodes-is-approved.
21. Endesa. *New air pollution protocol for Madrid*. 2018 [cited 2019 May 8]; Available from: https://endesavehiculoelectrico.com/en/espanol-nuevo-protocolo-de-contaminacion-de-madrid/.
22. Hansen, P.G. and A.M.J.E.J.o.R.R. Jespersen, *Nudge and the manipulation of choice: A framework for the responsible use of the nudge approach to behaviour change in public policy.* 2013. **4**(1): p. 3-28.

23. Thaler, R.H. and C.R. Sunstein, *Nudge: improving decisions about health, wealth, and happiness*. HeinOnline.
24. Hausman, D.M. and B. Welch, *Debate: To nudge or not to nudge.* Journal of Political Philosophy, 2010. **18**(1): p. 123-136.
25. Sustein, C.R., *The ethics of nudging.* Yale J. on Reg., 2015. **32**: p. 413.
26. Hanks, A.S., et al., *Healthy convenience: nudging students toward healthier choices in the lunchroom.* Journal of Public Health, 2012. **34**(3): p. 370-376.
27. Gosnell, G.K., J.A. List, and R. Metcalfe, *A new approach to an age-old problem: Solving externalities by incenting workers directly*. 2016, National Bureau of Economic Research.
28. Duflo, E., M. Kremer, and J.J.A.E.R. Robinson, *Nudging farmers to use fertilizer: Theory and experimental evidence from Kenya.* 2011. **101**(6): p. 2350-90.
29. Weinmann, M., C. Schneider, and J. vom Brocke, *Digital nudging.* Business & Information Systems Engineering, 2016. **58**(6): p. 433-436.
30. Schneider, C., M. Weinmann, and J. vom Brocke, *Digital Nudging–Guiding Choices by Using Interface Design.* Schneider, C., Weinmann, M., and vom Brocke, J.(2018). Digital Nudging–Guiding Choices by Using Interface Design, Communications of the ACM, 2017. **61**(7): p. 67-73.
31. Schneider, C., M. Weinmann, and J. vom Brocke, *Digital Nudging: Guiding Online User Choices through Interface Design.* Communications of the Acm, 2018. **61**(7): p. 67-73.
32. Rodriguez, J., G. Piccoli, and M. Bartosiak, *Nudging the Classroom: Designing a Socio-Technical Artifact to Reduce Academic Procrastination.*
33. Weiser, M., *What is Pervasive Computing?* 2001.
34. Silberschatz, A., H.F. Korth, and S. Sudarshan, *Database system concepts*. Vol. 4. 1997: McGraw-Hill New York.
35. SIR, V., *RELATIONAL DATABASE MANAGEMENT SYSTEM….* 1986.
36. Eder, J. and W. Liebhart, *Workflow transactions.* Workflow Handbook, 1997: p. 195-202.
37. Oussous, A., et al. *Comparison and classification of nosql databases for big data*. in *Proceedings of International Conference on Big Data, Cloud and Applications*. 2015.
38. Cáceres, J., et al., *Service scalability over the cloud*, in *Handbook of Cloud Computing*. 2010, Springer. p. 357-377.
39. Mohamed, M.A., O.G. Altrafi, and M.O. Ismail, *Relational vs. nosql databases: A survey.* International Journal of Computer and Information Technology, 2014. **3**(03): p. 598-601.
40. Gordon, K.J.I., *What is Big Data?* 2013. **55**(3): p. 12-13.
41. Chen, M., et al., *Big data: A survey.* 2014. **19**(2): p. 171-209.
42. Mell, P. and T. Grance, *The NIST definition of cloud computing.* 2011.
43. Buyya, R., et al., *Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility.* 2009. **25**(6): p. 599-616.
44. Grolinger, K., et al., *Data management in cloud environments: NoSQL and NewSQL data stores.* 2013. **2**(1): p. 22.
45. Gilbert, S. and N.J.A.S.N. Lynch, *Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services.* 2002. **33**(2): p. 51-59.
46. Grolinger, K., et al., *Data management in cloud environments: NoSQL and NewSQL data stores.* Journal of Cloud Computing: advances, systems and applications, 2013. **2**(1): p. 22.
47. Chandra, D.G., *BASE analysis of NoSQL database.* Future Generation Computer Systems, 2015. **52**: p. 13-21.
48. Hecht, R. and S. Jablonski. *NoSQL evaluation: A use case oriented survey*. in *2011 International Conference on Cloud and Service Computing*. 2011. IEEE.
49. Chang, F., et al., *Bigtable: A distributed storage system for structured data.* ACM Transactions on Computer Systems (TOCS), 2008. **26**(2): p. 4.
50. Mohan, C. *History repeats itself: sensible and NonsenSQL aspects of the NoSQL hoopla*. in *Proceedings of the 16th International Conference on Extending Database Technology*. 2013. ACM.
51. Pavlo, A. and M.J.A.S.R. Aslett, *What's really new with NewSQL?* 2016. **45**(2): p. 45-55.

52. Dragojević, A., et al. *No compromises: distributed transactions with consistency, availability, and performance*. in *Proceedings of the 25th Symposium on Operating Systems Principles*. 2015. ACM.
53. Inmon, W.H.J.P.T.T., *What is a data warehouse.* 1995. **1**(1).
54. Chaudhuri, S., B. Ding, and S. Kandula. *Approximate query processing: No silver bullet*. in *Proceedings of the 2017 ACM International Conference on Management of Data*. 2017. ACM.
55. Cattell, R., *Scalable SQL and NoSQL data stores.* Acm Sigmod Record, 2011. **39**(4): p. 12-27.
56. Carlson, J.L., *Redis in action*. 2013: Manning Publications Co.
57. team, M. *Design the infrastructure persistence layer*. 2018 [cited 2019 May 1]; Available from: https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design.
58. team, M. *The Repository Pattern*. 2010 [cited 2019 May 1]; Available from: https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff649690(v=pandp.10).
59. Fowler, M. *Unit of Work*. 2010 [cited 2019 May 1]; Available from: https://martinfowler.com/eaaCatalog/unitOfWork.html.
60. Karia, B. *A quick intro to Dependency Injection: what it is, and when to use it*. 2018 [cited 2019 May 1]; Available from: https://medium.freecodecamp.org/a-quick-intro-to-dependency-injection-what-it-is-and-when-to-use-it-7578c84fa88f.
61. Crusoveanu, L. *Intro to Inversion of Control and Dependency Injection with Spring*. 2019 [cited 2019 May 1]; Available from: https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring.
62. Alexander Wolny, e.a. *Mobility Choices App*. 2018 [cited 2019 May 7]; Available from: https://www.interreg.org/aktuell/mobility-choices-app.
63. Froehlich, J., et al. *UbiGreen: investigating a mobile tool for tracking and supporting green transportation habits*. in *Proceedings of the sigchi conference on human factors in computing systems*. 2009. ACM.
64. Jana Strozinsky, e.a. *A better day the 100 way*. 2017 [cited 2019 May 7]; Available from: https://eingutertag.org/en/.
65. Prof. Dr. Edith Maier, e.a. *E-Nudging – Motivationshilfe in der Prävention und im Umgang mit chronischen Erkrankungen im Alltag.* 2015 [cited 2019 May 7]; Available from: https://www.fhsg.ch/de/publikationen/publication/e-nudging-motivationshilfe-in-der-praevention-und-im-umgang-mit-chronischen-erkrankungen-im-alltag/.
66. Microsoft. *Visual Studio 2019*. 2019 [cited 2019 May 1]; Available from: https://visualstudio.microsoft.com/vs/.
67. team, M. *C# Guide*. 2018 [cited 2019 May 1]; Available from: https://docs.microsoft.com/en-us/dotnet/csharp/.
68. team, N. *nuget*. 2019 [cited 2019 May 2]; Available from: https://www.nuget.org/.
69. team, R. *RestSharp*. 2011 [cited 2019 May 12]; Available from: http://restsharp.org/.
70. Medley, J. *Web Push Notifications: Timely, Relevant, and Precise* 2019 [cited 2019 May 7]; Available from: https://developers.google.com/web/fundamentals/push-notifications/.
71. team, M. *About .NET Core*. 2018 [cited 2019 May 1]; Available from: https://docs.microsoft.com/en-us/dotnet/core/about.
72. team, M. *Entity Framework Core*. 2016 [cited 2019 May 1]; Available from: https://docs.microsoft.com/en-us/ef/core/.
73. team, O. *Oracle Database 18c*. 2019 [cited 2019 May 2]; Available from: https://www.oracle.com/technetwork/database/enterprise-edition/downloads/oracle18c-windows-180000-5066774.html.
74. w3schools. *SQL Tutorial*. 2019 [cited 2019 1 May]; Available from: https://www.w3schools.com/sql/sql_intro.asp.
75. Microsoft. *What you'll love about SQL Server 2017*. 2017 [cited 2019 May 1]; Available from: https://www.microsoft.com/en-us/sql-server/sql-server-2017.

76. Microsoft. *Language Integrated Query (LINQ)*. 2017 [cited 2019 May 1]; Available from: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/.
77. al., S.S.e. *Redis*. 2015 [cited 2019 May 1]; Available from: https://redis.io/.
78. team, T. *TypeScript*. 2019 [cited 2019 May 1]; Available from: https://www.typescriptlang.org/.
79. team, A. *Angular*. 2019 [cited 2019 May 1]; Available from: https://angular.io/.
80. team, n. *About npm*. 2018 [cited 2019 May 2]; Available from: https://docs.npmjs.com/about-npm/.
81. team, w. *Webpack*. 2016 [cited 2019 May 2]; Available from: https://webpack.js.org/concepts/.
82. W3C. *HTML 5.2*. 2017 [cited 2019 May 1]; Available from: https://www.w3.org/TR/html52/.
83. Mozilla. *CSS: Cascading Style Sheets*. 2019 [cited 2019 May 1]; Available from: https://developer.mozilla.org/en-US/docs/Web/CSS.
84. Park, T. *Bootswatch*. 2019 [cited 2019 May 1]; Available from: https://bootswatch.com/.
85. team, B. *Bootstrap*. 2019 [cited 2019 May 1]; Available from: https://getbootstrap.com/.
86. Microsoft. *Visual Studio Code*. 2019 [cited 2019 May 1]; Available from: https://code.visualstudio.com/.
87. Microsoft. *SQL Server Management Studio (SSMS)*. 2019 [cited 2019 May 1]; Available from: https://docs.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms?view=sql-server-2017.
88. Oracle. *SQL Developer*. 2019 [cited 2019 May 1]; Available from: https://www.oracle.com/database/technologies/appdev/sql-developer.html.
89. Jens Lorenz, e.a. *Noteapad ++*. 2019 [cited 2019 May 1]; Available from: https://notepad-plus-plus.org/.
90. Team, S. *SQL Database Modeler*. 2019 [cited 2019 May 1]; Available from: https://sqldbm.com/.
91. team, A. *AWS*. 2019 [cited 2019 May 7]; Available from: https://aws.amazon.com/?nc2=h_lg.
92. Ltd., J. *Draw.io*. 2012 [cited 2019 May 12]; Available from: https://www.draw.io/.
93. team, A. *AccuWeather APIs*. 2017 [cited 2019 May 12]; Available from: https://developer.accuweather.com/apis.
94. team, T. *Troms fylkestraffik*. 2019 [cited 2019 May 12]; Available from: https://www.tromskortet.no/?lang=no_NO.
95. team, G. *Directions API* 2019 [cited 2019 May 2]; Available from: https://developers.google.com/maps/documentation/directions/start.
96. team, G. *Calendar API*. 2018 [cited 2019 May 12]; Available from: https://developers.google.com/calendar/overview.
97. Oracle. *https://www.nuget.org/packages/Oracle.ManagedDataAccess/*. 2019 [cited 2019 May 14]; Available from: https://www.nuget.org/packages/Oracle.ManagedDataAccess/.
98. Rowan Miller, e.a. *Querying Data*. 2016 [cited 2019 May 7]; Available from: https://docs.microsoft.com/en-us/ef/core/querying/index.
99. atifaziz. *NCrontab: Crontab for .NET*. 2019 [cited 2019 May 13]; Available from: https://github.com/atifaziz/NCrontab.
100. Joe Stack, e.a. *APPROX_COUNT_DISTINCT (Transact-SQL)*. 2019 [cited 2019 May 7]; Available from: https://docs.microsoft.com/en-us/sql/t-sql/functions/approx-count-distinct-transact-sql?view=sql-server-2017.
101. Laker, K. *Is an approximate answer just plain wrong?* 2016 [cited 2019 May 7]; Available from: https://blogs.oracle.com/datawarehousing/is-an-approximate-answer-just-plain-wrong.
102. Hall, T. *APPROX_COUNT_DISTINCT : Quick Distinct Count in Oracle Database 12cR1 (12.1.0.2)*. 2014 [cited 2019 May 13]; Available from: https://oracle-base.com/articles/12c/approx-count-distinct-12cr1.

103. Mike Ray, e.a. *What's new in SQL Server 2019 preview*. 2019, April 23 [cited 2019 May 1]; Available from: https://docs.microsoft.com/en-us/sql/sql-server/what-s-new-in-sql-server-ver15?view=sqlallproducts-allversions.
104. Microsoft. *Database Providers*. 2018 [cited 2019 May 1]; Available from: https://docs.microsoft.com/en-us/ef/core/providers/.