Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Kristian Bortne

Comparison between RDBMS and NoSQL for Aviation Data

Master's thesis in Master in Informatics
Supervisor: Svein Erik Bratsberg
June 2019

NTNU
Norwegian University of Science and Technology

AVINOR

Abstract

Context: Databases is essential in any software product nowadays. The performance of databases might impact the product severely in terms of usage. It is therefore crucial that the performance of a chosen database is as good as it can get. New technologies and databases are developed at a rapid pace, and the choice of database can be a difficult decision. It is therefore needed to make comparisons between databases to see how each of them performs.

Objectives: This thesis will look at how one of the databases at Avinor could be used, the importance of indexes on databases, characteristics of NoSQL databases and its features, and a performance experiment between an RDBMS and a NoSQL database.

Methods: The research methods that have been used is a literature review based on published papers, reviews and articles found on different sites and an experiment. The experiment is structured with defined cases and queries, and executed on both an RDBMS and MongoDB. The measurement of performance is based on the result and feedback from their query optimizer.

Result: It was found out that a document-based database would be most suited as a NoSQL alternative, and MongoDB was chosen due to various reasons. Based on the experiment, MongoDB performed better than the RDBMS whenever the RDBMS required *joins* to fetch the result. When not, the RDBMS performed better than MongoDB.

Conclusion: MongoDB has a few unique features compared to a relational database that would be beneficial, like flexible schema structure, auto-scaling, and replication across many nodes. However, since the solution is already in an RDBMS and the performance gap is minimal between them, it would not be recommended to switch to a NoSQL solution as of now. In the future, with increased data size, this might be a more viable option.

Keywords: Database, RDBMS, NoSQL, MongoDB, Performance Comparison.

Sammendrag

Kontekst: Databaser er essensielle i nesten alle programvareprodukter i dag. Ytelsen på databaser kan påvirke produktet i stor grad når det kommer til bruk. Det er derfor avgjørende at ytelsen er så god som det er mulig å få den. Nye teknologier og databaser blir utviklet i et forrykene tempo, og valg av database kan være et vanskelig valg. Det er derfor nødvendig å gjøre sammenligninger mellom dem for å se hvordan ytelsen er.

Målsetting: Denne oppgaven vil se på bruksområdet for en av databasene til Avinor, betydningen av indekser på databaser, egenskaper ved NoSQL databaser og et resultateksperiment mellom en relasjondatabase og en NoSQL database.

Metode: Forskningsmetodene som er blitt brukt i denne oppgaven er litteraturgjennomgang basert på publiserte papirer, artikler som finnes på forskjellige nettsteder og et eksperiment. Eksperimentet er bygget opp med lik data og spørringer for å teste ytelsen på relasjonsdatabasen og MongoDB. Resultatet er hentet ut fra spørringsoptimalisatoren til hver av databaseteknologiene.

Resultat: Det ble funnet ut at en dokumentbasert database ville være best egnet som et NoSQL alternative. Her ble MongoDB valgt basert på diverse egenskaper. Basert på eksperimentet, var ytelsen på MongoDB bedre enn ved relasjonsdatabasen, hvor man møtte bruke *join* i en relasjondatabase. For spørringer hvor dette ikke var nødvendig, var ytelsen bedre for relasjonsdatabasen enn for MongoDB.

Konklusjon: MongoDB har noen unike egenskaper som ville vært gunstig i forhold til en relasjonsdatabase, som for eksempel fleksibel skjemastruktur, automatisk skalering og replikering på tvers av mange noder. Siden løsningen allerede eksisterer i en relasjonsdatabase og ytelsesforskjellen er minimal, anbefales det ikke å bytte til en NoSQL løsning på nåværende tidspunkt. I fremtiden, med økt datamengde, kan MongoDB være et bedre alternativ enn dagens løsning.

Nøkkelord: Database, RDBMS, NoSQL, MongoDB, Sammenligning av Ytelse.

Preface

This thesis is the final delivery in the Informatics program at the Department of Computer and Information Science (IDI), at the Norwegian University of Science and Technology (NTNU) in Trondheim. The work was completed during August 2018 and June 2019, under supervision of Svein Erik Bratsberg.

Acknowledgements

I want to thank Svein Erik Bratsberg for the support, assistance and discussion of ideas throughout this project. I would also like to thank Avinor for their support and guidance during this project and for giving me this opportunity. A special thanks to Anette Johnsrud, Morten Banzon, Ole Nymoen and Flemming Hølvold at Avinor for taking their time to answer questions and have meetings about their daily business and technical operations.

Table of Contents

List of Figures

Abbreviations

Technical terms

RDBMS	=	Relational database management system
ACID	=	Atomicity, Consistency, Isolation, Durability
BASE	=	Basically Available, Soft State, Eventual consistency
OLAP	=	Online Analytical Processing
OLTP	=	Online Transaction Processing

Aviation terms

AODB	=	Airport Operational Database
TRDB	=	Transaction Reporting Database
RDB	=	Reporting Database
ALTi	=	Avinor Air Traffic information / Avinor lufttrafikkinformasjon
AOBT	=	Actual Off-Block Time
ATOT	=	Actual Take Off Time
ETOT	=	Estimated Take Off Time
ELDT	=	Estimated Landing Time
ETFMS	=	Enhanced Tactical Flow Management System
ECAC	=	European Civil Aviation Conference

Chapter 1

Introduction

This first chapter presents an overview of the background and the motivation for doing this thesis. A brief overview of the background is needed to understand the main goal, and Chapter 2 will have a more in-depth focus on the background theory. The main goal and research questions will then be presented to the reader. Then the related work and literature review will be presented, before the contribution and thesis structure is presented at the end.

1.1 Background

Avinor is a state-owned company that own, operate and develop a national network of airports for the civilian sector and joint air navigation services for the civilian and military sectors. They have the responsibility for 44 airports in Norway, making them one of the most hyper-connected aviation networks in the world. For the last few years, the commercial flight movement has decreased by a few percents, but the amount of passengers and cargo has increased significantly in Norway [9]. In 2018, there were over 54 million people that travelled through the airports of Avinor. In addition to this, there was just below 800.000 combined departures and arrivals in terms of air traffic movement[10] [Norwegian, [11]].

Even with the high amount of aircraft movement and a high number of passengers, for Norway to be, some of the biggest airports in Norway have the highest on-time percentages in the world according to OAG Aviation Worldwide [12]. Stavanger, Bergen, and Trondheim are in the top 12 categories for smaller airports, while Oslo is rank 16 in the category for larger airports, based on worldwide statistics for measuring on-time performance.

Avinor is working continuously on improvements that can be done in their operations to reduce delays, increase passenger satisfaction and increase efficiency. More and more data is collected, and this data needs to be analyzed in order to give more insight and to improve. The data can be from aircraft movement, the pattern of actions for passengers or internal systems like baggage handling, fueling, docking, etc.

This thesis will take on one of their databases called TRDB, which holds data for all flights arriving or departures from an Avinor airport and all flights that are in Norwegian airspace. TRDB is one of several components that were to be delivered in a project for the acquisition of a new traffic information system a few years ago. It is a transaction database and is supposed to hold the complete history of every change that is made towards a flight. At this point, the database and the data are not in use since the data quality in TRDB does not satisfy production quality.

The database is already in an RDBMS, but the execution time of queries are not suitable for operational usage. The execution time for each query is usually above 60 seconds or even higher. This paper will look at how the database is constructed today and how it can be improved. It will contain an experiment with a NoSQL system, where a comparison will be made to look at the differences between an RDBMS and a NoSQL system, in performance and what characteristics that can be achieved.

1.2 Goals and Research Questions

Main goal: Improve TRDB concerning execution time and overall performance for user queries.

The main goal for this master thesis is to see how the TRDB database at Avinor can be improved. With the amount of data that is stored, the computation time should be heavily reduced, considering the extent of all data. A series of research question has been formed to cover the essential part of the main goal in order to give a conclusion at the end of the thesis.

Research question 1: How is TRDB used and what data is important for usage?

To be able to improve a database, it is important to know how the database is used, what kind of data that is being used and the frequency of usage. As of today, TRDB is not approved to use in production due to incorrectness in data and its quality. This is either due to incorrectness of data in AODB, the source of data, or the communication between them. Since there are no default queries that are being run, the queries must be made in this thesis based upon use cases that Avinor has mentioned. Making queries for use cases tells what kind of data is most important and what should be focused on.

Research question 2: How much can TRDB be improved in a RDBMS with indexes?

Avinor is aware that there are no indexes in the database and that this has a significant impact on the execution time of queries. There has, however, been no analysis to look at how much this can impact the performance and how important it is. Indexes will be added to the database in RDBMS and a comparison before and after will be done.

Research question 3: Which NoSQL system is most suited to replace the RDBMS of TRDB?

To be able to tell which NoSQL systems is an alternative to replace the RDBMS of TRDB, a review of four different NoSQL categories will be done. These four categories are;

Document-based, Key-value, Graph and Column Family databases. This will highlight their advantages against each other and what they are most suited for. A decision will be made based on their features and a suitable candidate from that group will be selected for further experiment.

Research question 4: Can a NoSQL system of TRDB be as efficient or better than a solution in RDBMS?

The goal here is to make a mock-up database of TRDB in NoSQL and populate it with the same kind of data. It will then be possible to make queries for each technology, look at the result and find out which one has the highest performance in terms of execution time. Since the data is mostly going to be used for searches, it is important that it is as fast as possible, since more and more data is being collected and needs to be analyzed.

1.3 Research Method

The research method for this project will be divided into two separate parts. The first part will be based on a literature review in order to see what NoSQL technologies would be most suited. This involves looking at the structure of the data and its use cases, and see what features from NoSQL databases that would be beneficial for development. The goal here is to limit the number of different technologies and only have one candidate that will be used forward into the next part.

The second part will be an experiment between the solution today in an RDBMS and a alternative in NoSQL. The reason for experimenting is to do an analysis of the two technologies and find out who has the highest performance. In the experiment, there will be several different cases and tests, making sure that the two systems will be tested under different circumstances. After the analysis, it will be possible to say how they compare to each other and which one is the best alternative.

1.4 Related work

Comparison between database systems is nothing new and has been done multiple times. RDBMS have been compared to each other, NoSQL systems to each other and comparison between RDBMS and NoSQL has been done. Papers [13], [14], [15], [16], [17], [18], [19] and [20] all try to compare different systems against each other. A wide variety of systems has been compared, but a different result is or may be presented. There are many variables when comparing systems, making the result varies. Measurement of performance is one of them. It is difficult to achieve the same method for measuring when it is two different systems. Data set is also one of the variables. The performance heavily depends on the data set, data structure and the usage of this data. Because of these variables, it is impossible to say that one system is better than the other in general. This is a part of the motivation for this thesis, to investigate the difference between two database systems for a real-life scenario.

There are no papers that are referring to the use of NoSQL systems in the aviation world. Numerous attempts at finding papers has been used, without much result. Sven

Hafner works with Airport IT Systems and Integration Architect, and has been working in the aviation industry for a while. He has written a blog about the possibility of using a NoSQL system for a database similar to TRDB. This is the closest source that is similar to this thesis. The posts about NoSQL is divided into multiple parts, and the series of these is not done. There is, therefore, no conclusion that can be drawn from the blog posts that has been written by Hafner [21][22].

1.5 Structured Literature Review Protocol

Most of the literature that was reviewed has been found through Google and Google Scholar. Google Scholar[23] is a search engine that returns results from scientific papers and books. By using papers and books as a reference, the credibility increases since it already has been review by other professors or professionals before being published. There is little to no information about databases in the aviation industry and nothing with NoSQL systems. Therefore a more general approach has been made due to this limitation in literature. Search words that was used for example; ”Comparison relation database and NoSQL”, ”NoSQL history database”, ”NoSQL transaction database”, ”NoSQL use cases”, etc. The literature has been read through with the research questions and the topic in mind.

Due to the lack of literature, a few web pages and blogs have also been used in this thesis. This is to supplement the papers that were found. Used web pages/ blogs have been evaluated more since it may not been review by others before publishing. In addition to this, documentation based on various systems has also been included.

1.6 Contributions

To my knowledge, this has not been tested out earlier in aviation. My thesis might help Avinor in further development and decision making of TRDB. This means how TRDB easily can be improved in the already existing solution and how new technology might contribute to better solutions. The thesis is written in cooperation with Avinor, but other actors in the aviation industry might also find the result relative to their interest.

1.7 Thesis Structure

To give insight over what will be discussed in the next chapters, any overview can be seen here:

- **Chapter 2 - Background Theory:** All basic theory is introduced and explained here. The first part will give an in-depth introduction to TRDB and how the system is connected to others. It describes the structure, usage, and storage of the database. The second part gives relevant theory about NoSQL systems in general. Many of the characteristics of a NoSQL system is explained here.

- **Chapter 3 - TRDB in RDBMS:** A few of the use cases of TRDB will be discussed to a great extent. We will look at what kind of usage is covered, how SQL queries can be structured to give the necessary data for the use case and for each query look at the execution time of the existing solution. Improvements will be done and result before and after will be presented to the reader.
- **Chapter 4 - NoSQL Technology:** The choice of NoSQL system will be taken, and the reason is explained. A look at the data structure and how it differentiates from the RDBMS is talked about. Queries for each use case will be created, and the result will be presented to the reader.
- **Chapter 5 - Comparison of Results:** Each of the database systems will be compared before and after any improvements were made to see how the improvement impacts the performance. Then the database systems are compared to each other in order to see which of them performs best. The chapter will also include a discussion of the result and its limitations.
- **Chapter 6 - Conclusion and Future Work:** Based on the result that was generated, a conclusion will be made to determine which of them would be most suited for usage at Avinor. The conclusion is based on both technological advantages and business operation. Future work is included at the end to give an insight into what should be focused on.

Chapter 2

Background Theory

2.1 TRDB Theory

In this chapter, the theoretical material will be presented to the reader. This is essential to get a deeper understanding of how TRDB is structured in Avinor and what the problems are. All information is based upon internal documents in Avinor that is not public and therefore lacks references.

TRDB stands for Transaction Reporting Database. There are two main purposes for this database in Avinor. The first one is to keep a complete historical record of all flight leg information. In the aviation world, a *flight leg* is a movement for a flight from point A to point B. A scheduled flight will be scheduled months or up to a year in advance of their take-off. The second purpose is to keep track of all of the changes that have been made, when it has been made and who made the change. From schedule point until take-off time, there might be many changes such as schedule time for take-off time, registration number, gate number and many more.

2.1.1 Sources of data

Avinor gets its data from multiple different sources. Looking at Figure 2.1, Chroma Updates and Chroma Partner Services are the primary sources of information. SITA and SSIM are part of what is called Chroma Updates. SITA and SSIM are sources outsize of Avinor that sends updates in the form of messages or files, which contains flight information.

Internal systems like docking and baggage system uses Chroma Partner Services indirectly to update information in AOBD. An example of this can be the docking system, telling when an airplane is actual in-block time, meaning that the airplane has arrived at its aircraft parking position.

In addition to these, information from Eurocontrol is also received. Eurocontrol is an organization in the EU that is responsible for controlling the European airspace and makes sure that it is safe to fly. Eurocontrol monitors every flight and sends information

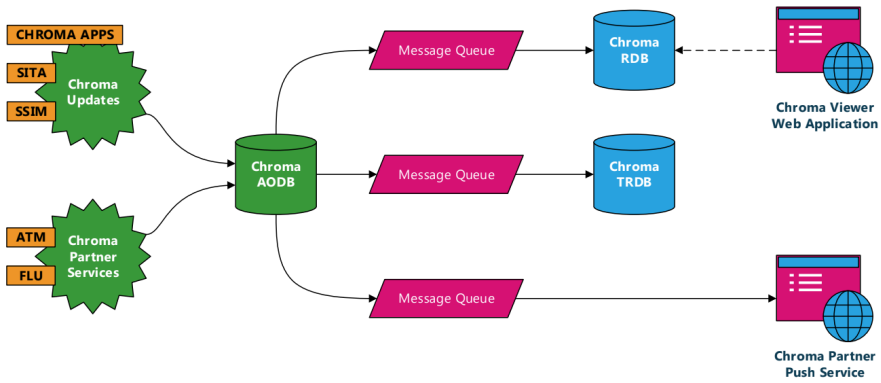


Figure 2.1: Overview of the database architecture. Source: Leidos

to airports. If a flight is going to arrive at Oslo Airport, Avinor will receive information about this flight underway its flight, such as estimated take off time (ETOT) and estimated landing time (ELDT), making it easier to consider inevitable delays.

In addition to getting information about flights that are arriving or departing on Avinor airports, it receives data from Eurocontrol about all aircraft in Norwegian airspace. There are multiple purposes for this like security and safety, but the data is used for tax calculation as well. All airlines need to pay taxes to countries, based on their route of travel and passenger number.

2.1.2 Storage of data

There are three different databases that stores data about flight leg. Chroma Airport Operational Database (AODB) is the heart of the information storage. This is the database that holds information about all of the flights that are going to an Avinor airport. In addition to AODB, TRDB and RDB (Reporting Database) was made. RDB contains only the latest flight leg information, and TRDB contains every change that has been gathered by AODB. There are two reasons for having three separate databases. The first reason is to reduce the load on the database AODB since this is the main storage of information. The second reason is not to compromise the security of the information that is in AODB.

The data from AODB to TRDB is being sent through a message queue asynchronously. This is done to avoid any synchronous blocking and improve the performance of AODB and throughput rate on TRDB. If the TRDB database goes down for maintenance or any other reasons, the queue will be filled and delivered to the database ones it is up. Depending on the length of the offline period and the activity, it may take a few hours/days to transfer all of the data to TRDB.

TRDB have until now reference as one database, but in practice, it consists of two similar parts. For every update that AODB gets from different sources, TRDB gets a message that is split into two parts, see Figure 2.2. The first part is the new state of AODB caused by the update, and the second part is the update that was received from various sources. For each update that is received, the state does not necessarily change. Sources

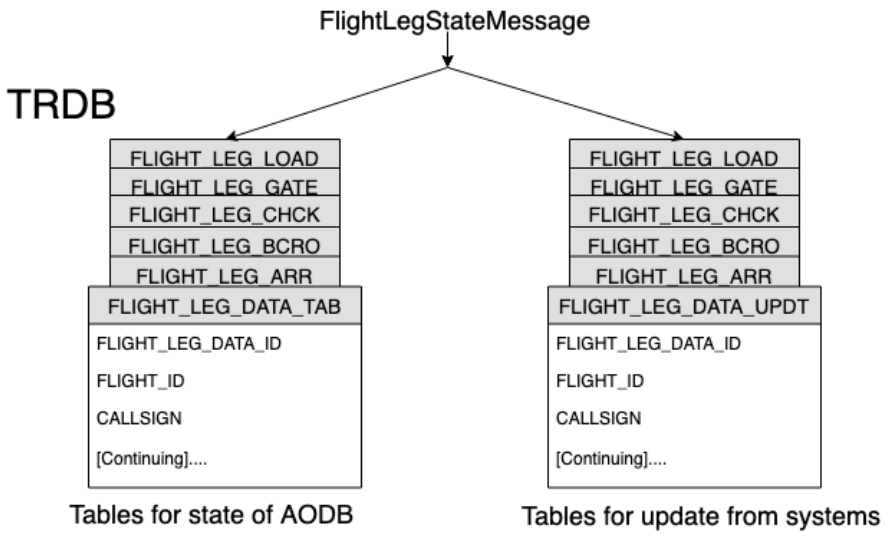


Figure 2.2: Separate parts of TRDB.

have different prioritization, meaning that AODB might ignore updates. If this were to happen, the update would still be sent to TRDB, but the state would be unchanged. This allows Avinor to have full control of their data and understand how and what updates that change data.

An example of a message can be seen in Appendix II. The message that is received is in XML format. In the first couple of lines, there is metadata like time stamps, event, and source, and it is defined under the tag *ambiHeader*. The next tag is *flightLegDataChange*. This is the update message, containing metadata and data that a system wants to update. The data in this tag is stored in the update tables of TRDB. The next tag in the XML message is *flightLegDataState*. This is the state of data that is being stored in AODB. The data from *flightLegDataState* is being stored in the state tables of TRDB, therefore containing all previous versions of data in AODB.

2.1.3 Matching of data

Whenever TRDB receives data, it has to do a matching to see which transaction is the newest. One of the tables in TRDB has a column named *LATEST_VERSION*. This field has to be updated for the previous state/update if a newer state/update is received. This is done to always know which state/update is the newest and therefore most relevant.

Usually, in a relational database, there would be a consistent key or index that could be used to see previous transactions. An example of this could be a bank and a customer. The customer has its own key/index, and it never changes. When checking if a transaction for a customer is the newest, the customer key/index can be used to find the previous transaction and update accordingly.

In TRDB, there is not such a consistent key. There is always a possibility that the data

changes for every update. Therefore, there is a sequence of different data fields it checks, to verify what is the correct previous transaction for a flight:

1. Flight ID.
2. Flight Departure Date.
3. Flight Departure Airport.
4. Flight Arrival Airport.
5. Schedule Off-Block Time (SOBT).
6. Schedule In-Block Time (SIBT).

If most of the data fields above matches the received data, it is a match. The time stamp is then checked to see which one is the newest, and then updates and inserts the data accordingly.

The difference from the received data, and the data in TRDB might be too big, thus not creating a match, even tho it is the same flight. If this happens, the received data will be inserted into the database, and the old data will not be updated. This means that there would be two existing records for the same flight, something that is not wanted. It is therefore crucial that the system can do such a match with the available data.

2.1.4 Usage of TRDB

As can be seen in Figure 2.1, there are no outgoing connections to TRDB. As of today, the database is not in use, but Avinor already has certain use cases that they want TRDB to achieve. These use cases can be split into six different categories:

- **Statistics:** Airport Council International and Statistisk Sentralbyrå are two different organizations that receive statistics about flight movement in Norway. These statistics are often aggregated to months or year. These statistics usually contain the number of flights in a period, the number of passengers, cargo, registration, callsign, type of aircraft, etc. An example of such statistics for Oslo Airport can be seen in Appendix I.
- **Analysis:** Time of landing is essential to be sure that handlers, people taking care of the airplane, is present when the airplane comes to the aircraft parking position. There is one source that generates the truth, and that is the docking system. This system tells when the plane has arrived at the allocated aircraft parking position. It is essential to know when an airplane will come to the aircraft parking position, but the truth is only generated ones the plane has parked. There are, therefore, many systems that try to estimate the landing time of all planes, which indicates when the plane will arrive at the gate. However, all of these systems that estimate the landing time can estimate different landing times. Based on the truth that the docking system generates and knowing the distance from the aircraft parking position to the runway, it is possible to determine which system gave the best-estimated landing time. It is then possible to rank all of the systems on accuracy and use the best system as a guideline.

- **Real-time Analysis:** Avinor is not able to generate or estimate all of the data that is necessary for a flight. Passenger number is one of these examples and is essential for doing tax calculation and crew handling. Airlines are supposed to send needed data for a flight within a given time frame. By doing real-time analysis, Avinor would be able to find out what kind of data is missing and take actions much earlier than they otherwise would.

- **Reporting:** Avinor is responsible for reporting to other European organizations like Eurocontrol. To Eurocontrol, they manually upload a report every month for all departures and arrivals. They are also responsible for reporting to other none Avinor airports in Norway about flights for departure and arrival that might impact that airport. This is a process that is done manually through email every day.

- **Tax Calculation:** All flights that enter the Norwegian airspace has to pay taxes depending on the type of aircraft and number of passengers. Today they have a system for this, but this will eventually be phased out, and be replaced by a new system that uses TRDB as the data source.

- **Passenger Prediction:** Beontra is a system for passenger prediction to give a stable base for a continuous, automated forecast of passenger streams in a terminal. This systems merge data from various sources and makes an estimation based on the data. TRDB, tracking systems, weather predictions and airline's booking data is just an example of data that is used for this system. TRDB is not responsible for processing data for this use case but should return relevant flights.

2.1.5 ER-model

In Figure 2.3, an ER-model can be seen of TRDB. The main table consists of a primary key named FLIGHT_LEG_DATA_ID. This key is passed on to every child as a foreign key. All children is a [N:1], meaning that there can exist many children for each parent. In addition to the primary key in the main table, there is a total of 218 columns. The 12 children tables consist of 4 to 15 columns. Each table consists of a primary key and a foreign key, which is the primary key of the main table.

2.1.6 Development of TRDB

TRDB is a component of a huge project called ALTi (Avinor Air Traffic information) at Avinor [Norwegian [24]]. Development of this project is done by a third-party company that specializes in information systems to airports. The project was started seven years ago and was formally concluded at the end of 2018. Due to the scope of the project, it has been delayed multiple times, and certain parts of the project has cut due to time constrictions. Development by the third-party company consist now of fixes from previous development and updates in agreement with Avinor.

Avinor is not able to make changes since the project is maintained and developed by the third-party company. Wanted changes from Avinor, needs to be requested in order to be committed. This process is time-consuming, and it can take weeks or months before

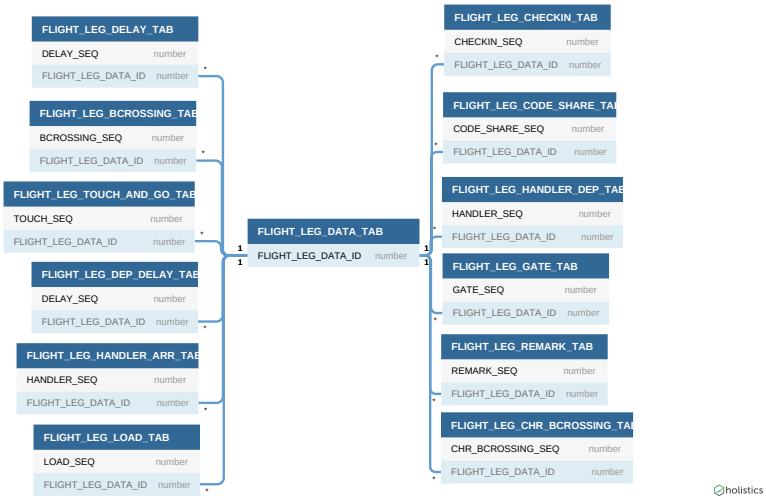


Figure 2.3: ER-model over the TRDB database. All columns are not added.

Name of tables	Total size in B	Total size in GB
FLIGHT_LEG_*_TAB	10812864917	10,7
FLIGHT_LEG_*_UPDT	5505112911	5,13

Table 2.1: Total storage of TRDB.

specific changes are pushed to production. In this age and time, it is essential to change rapidly, if needed, to keep evolving and keep up with user and customer satisfaction.

As mentioned earlier, TRDB is not in use due to incorrectness in data quality. This is the cause that leads to colossal execution time since the usage has not been analyzed before, and no improvements have been done or analyzed in order to improve the existing solution.

2.1.7 Physical storage and configuration of TRDB

As mentioned earlier, TRDB is divided into separate parts, but within the same database. One part takes care of all updates that are received, and the other part takes care of the state of data that has arrived from AODB. The total size of each part can be seen in Table 2.1 below. For further details on physical storage, see Appendix III. The data was gathered through a user interface that calculates the average length of each row and the number of rows. It is important to keep in mind that the RDBMS has a few techniques to reduce the storage space used. This is, for example, achieved through various data types. It is unknown if the RDBMS uses additional compression when listing the storage used.

Avinor themselves has done estimation of the update frequency that the system should be able to sustain. All Partner Service operations should be able manage a sustained throughput of at least five updates per second. Avinor estimates that TRDB will receive up

to 100.000 updates each day and that each update will be stored as a row in the database.

As of today, the size of the database is not an issue. The scalability of the existing solution might be an issue in the future and that the need for a new replacement will be more significant. This is a problem that will be discussed to a greater extent later on in this thesis.

2.1.8 Experimental setup of TRDB

A local and closed version of TRDB were created for this thesis, instead of using the test server of Avinor that is running TRDB. The reason for creating a local version of the database was to have full control over user roles and rights to the database, control the hardware and to make sure that the database was running with a few background processes as possible.

The experimental setup that is running TRDB locally:

- CPU: Intel Core i7-4790 CPU @ 3.60GHz
- RAM: 12 GB
- GPU: NVIDIA GeForce GTX 750 Ti 8127MB
- HDD: Seagate Desktop HDD 2TB 64MB Cache SATA 6.0Gb/s 3.5 7200RPM
- Operating system: Windows 10 Pro 64-bit

The hardware that is running TRDB locally is running on a desktop, not a server. Servers usually have better specs, and it is, therefore, fair to assume that the server at Avinor has more CPU cores and more memory available, thus making the hardware above not representable for the server that is hosting TRDB at Avinor.

This hardware will be used throughout this thesis for all experiments.

2.2 NoSQL Technology

NoSQL technology has been growing rapidly in the last century. Many characteristics define NoSQL technology and make them different from traditional relational database systems. To have a better understanding of NoSQL technology and what it can offer in general, a look at some of the key characteristics will be presented.

2.2.1 Scaling

Traditional relationship database systems naturally scale in a vertical direction. This involves adding more physical resources to the underlying server that is hosting the database, whenever an upgrade is needed. Physical resources would typically be more or better CPU, memory and storage [25]. This method has its advantages in its simplicity since it is easier to implement and administer a single server instead of a cluster of servers. The same code that is connected to the database is also the same and needs no changes.

There is, however, a few drawbacks with this method. A server can only be so big, and at one point, a relational database would not be sufficient if it gets too big. There is a cap on the hardware, and the high-end hardware is also costly, making it a costly process. Some of the relationship database systems use licenses, and customers would normally pay for these licenses based on the number of cores that the database is running, meaning that hardware is not the only pricey upgrade that is required [26].

Most NoSQL systems use horizontal scaling, instead of vertical scaling. Horizontal scaling refers to scaling by adding new processors with their separate disk, called a node. A NoSQL system is usually built up with many nodes that are connected. Each node contains only a subset of all the data that is stored in the database. *Sharding* is referred to the process of assigning data to each node, something that is automatically done in a few NoSQL systems. [27]. By scaling horizontal, a nearly linear scaling can be achieved [28]. When using horizontal scaling, commodity servers/hardware can be used to reduce the price. It is often cheaper to buy a new node than to upgrade an already existing node with the newest hardware.

As well as with vertical scaling, there are disadvantages. The complexity of horizontal scaling can lead to more bugs in the code and infrastructure is more complex. Also, the license fee can be higher for each node, the cost of running multiple nodes is higher, and the networking cost is higher.

2.2.2 Replication

Replication of data and fault tolerance is generally easier to achieve and manage due to the multiple nodes [26]. There are mainly two methods for handling replication in a distributed system. The first method is called Master-Slave replication. In a given cluster, one of the nodes is designated the role as a master. The master node is the only node this is allowed to write and update data and is, therefore, an authoritative source. The master node will propagate all of the updates to the slave nodes, which is the rest of the cluster. All of the slave nodes can read data but is not allowed to write data. This method avoids any conflicts on data since there is only one source for updates. If the master node goes down, a slave will be appointed as the new master quickly.

Peer-to-Peer replication is the second method. All nodes in the cluster can read and write data. If a node receives an update, that node has to propagate the data to all of the other nodes[29].

In addition to the two methods, there are two methods for sending data. Synchronous means that the update is propagated to all of the nodes before returning success to the user. By using an asynchronous update, success is returned to the user before the data is replicated on other nodes. This means that the system can give the ability to tune the trade-off between consistency and durability for performance for what fits the risk/performance [30].

2.2.3 CAP theorem

In 2000, Eric Brewer held a keynote talk about distributed systems and his experience with the development of distributed databases [Towards Robust Distributed systems [31]]. In this theorem, Brewer stated that it is not possible to have tolerance towards network

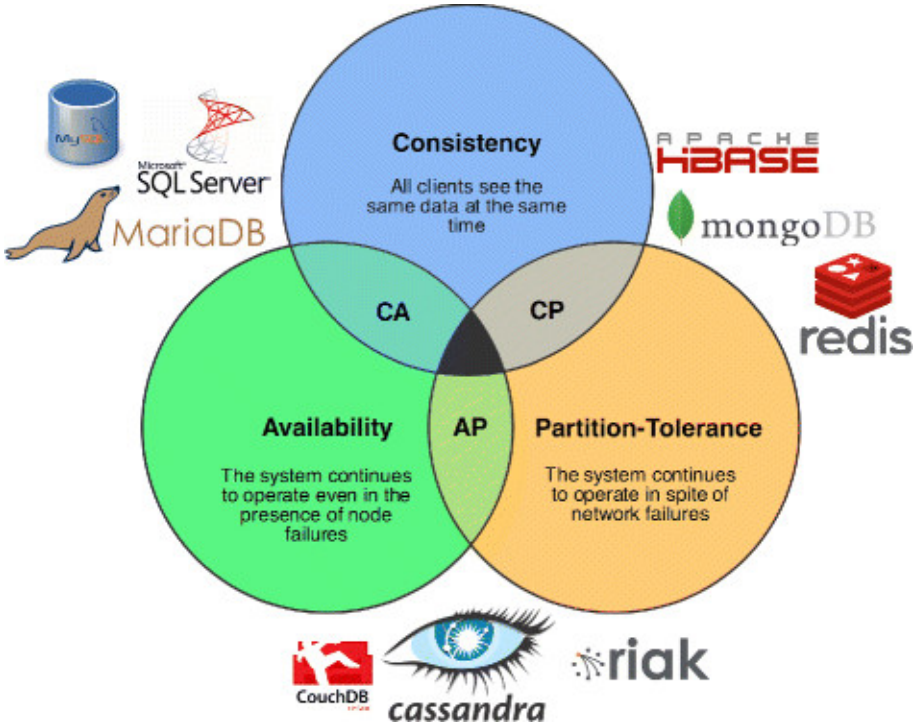


Figure 2.4: Visualization of the CAP theorem. Source: [1].

partition, consistency, and availability at the same time, see Figure 2.4. It was only possible to have two of the three features at a time. ”Consistency feature guarantees that all clients of a data management system have the same view of data. Availability assures that all clients can always read and write. Finally, partition tolerance ensures that the system works well with data that is distributed across physical network partitions” [27].

One of the characteristics of a NoSQL system is that it is distributed over many physical network partitions. Based on the theorem from Brewer, NoSQL systems must, therefore, choose between Consistency (ACID) or Availability (BASE). The right choice of the feature depends on the usage of the system and must be decided by the developers.

In 2012, Eric Brewer wrote a follow-up report on the CAP theorem that he made 12 years ago. He stated that designers and researchers had sometimes abused the CAP theorem and that it was misleading due to oversimplification [32]. According to Brewer, there are many levels and nuances between the three features and could not be represented in binary. In other words, only two of the features would be guaranteed. System designers should not blindly sacrifice availability or consistency when partitions exist. Instead, a trade-off between consistency and availability could be chosen. This means that two of the features can be guaranteed at any given time. The last feature is not guaranteed but is still very much present in the system.

2.2.4 Flexible schema structure

A relational database is usually constructed based upon a relational data model. Data is stored in tables, which consist of rows and columns and is never repeated in the database. This reduces the amount of data that is stored in the database and is more suited for data retrieval. The schema is very strict and defines what data type of data is stored in the database. Updating these schemes is a well-managed activity, time-consuming and the application has to go through rigorous software testing [27].

NoSQL schema, however, develops over time [33]. The schema is flexible meaning that the database system stores whatever is received as long as it follows the correct format. At this time and age where data is created rapidly and technology changes, this feature would make it easier to add new data to the storage without changing the software or the structure of the database. The downside of this feature is that developers might lose control and overview over what kind of data is stored in the database, since there is no schema to follow.

Data that is stored in NoSQL databases usually tend to use more space and storage then it would be compared to an RDBMS system. This is because more metadata is stored in the database in order to keep track and structure of the data. An example of this would be to add the key for each value, which can be seen in document-based databases. Storage usually is not a problem anymore since the price of buying hardware storage is relatively cheap compared to earlier years [34].

2.2.5 Maturity

Most of the relational database systems have been around for a long time and is well known in the tech industry. They have proven to be both robust and functional and trusted by most developers for decades. NoSQL has only been around for a few years, even tho it was first mentioned by Carlo Strozzi back in 1998 [35]. Many of the NoSQL alternatives have many key features yet to be implemented, but the development of these features happens rapidly. NoSQL fulfil a need in the tech space but is not as mature as relational database and CIO of big companies might find the maturity of RDBMS more reassuring [36].

Support

Big companies and enterprises need support for their infrastructure in case a critical failure happens to their systems. Many of the RDBMS vendors offer this high level of enterprise support. At the beginning of NoSQL history, most of them were open-source and would not be able to give support to [36]. In 2019, there are a few NoSQL systems that offer enterprise support like MongoDB [37], and more of them are probably to come in the future.

Expertise

RDBMS and SQL language are known by most developers around the world, which is estimated to be around 26 million people in 2019, according to Evans Data Corporation [38]. The SQL language and its syntax are commonly shared among all RDBMS systems.

This means that it is relatively easy to change RDBMS since the similarity is close. NoSQL systems, however, are under constant changes and developers are in a learning mode in most cases [36]. It is harder and more difficult to find developers with knowledge about NoSQL than standard SQL. If a company is to change from RDBMS to NoSQL system, it is important that there are developers who know about maintaining and using the database. Due to the lack of knowledge and expertise in NoSQL, it is an easier and safer choice to choose RDBMS for companies, even if NoSQL is beneficial.

Chapter 3

TRDB in RDBMS

To understand how to improve the already existing system, it is important to know how the data will be used. Since the database today is not in use regularly, there are no default queries that are being executed. In this chapter, an in-depth look at how the use cases and the queries can be structured, along with the result of the queries before and after indexing.

3.1 Use cases

In this section, a closer look will be presented at the wanted use cases of TRDB, what data is needed, and how the queries might be structured to give the correct data. All of the use cases that are talked about in this chapter have been briefly talked about in Section 2.1.4.

3.1.1 Statistics

As mentioned in Section 2.1.4, Airport Council International[39] and Statistics Norway [40][41] are two different organizations that receives statistics from Avinor on a monthly basis. However, these organizations are not the only one using the statistics. Avinor themselves publish statistics to the public on their website[10], with details of calculation[11]. In addition to these standard cases, there are occasional requests from municipalities, on passenger movement to and from airports. This data is used to plan infrastructure and scheduling for public transport. Another example of usage would be to determine air traffic movement around holidays and have the proper amount of staffing at the appropriate time.

The use of statistics is broad, but in this thesis, the focus will be on the statistics that are being sent to Airport Council International, which can be seen in Appendix I.

The data that is needed to generate the report is divided into three categories; Aircraft Movement, Commercial Passengers, and Cargo. To get the necessary data for each category, a SQL query has been made for each of those. One of the examples can be seen below:

```
select
    Q1.FLIGHT_SERVICE_TYPE_IATA, Q1.DEPARTURE_AIRPORT_IATA as AIRPORT,
    sum(Q1.movement + Q2.movement) as TOTALMOVEMENT
from (select f.FLIGHT_SERVICE_TYPE_IATA, f.DEPARTURE_AIRPORT_IATA,
    count(*) as MOVEMENT
    from FLIGHT_LEG_DATA_TAB f
    where extract(month from f.FLIGHT_DEPARTURE_DATE) = extract(month from SYSDATE)
        and extract(year from f.FLIGHT_DEPARTURE_DATE) = extract(year from SYSDATE)
        and f.LATEST_VERSION = 'Y'
        and f.AOBT is not null
    group by f.FLIGHT_SERVICE_TYPE_IATA, f.DEPARTURE_AIRPORT_IATA) Q1
join (select f.FLIGHT_SERVICE_TYPE_IATA, f.ARRIVAL_AIRPORT_IATA,
    count(*) as MOVEMENT
    from FLIGHT_LEG_DATA_TAB f
    where extract(month from f.FLIGHT_DEPARTURE_DATE) = extract(month from SYSDATE)
        and extract(year from f.FLIGHT_DEPARTURE_DATE) = extract(year from SYSDATE)
        and f.LATEST_VERSION = 'Y'
        and f.AIBT is not null
    group by f.FLIGHT_SERVICE_TYPE_IATA, f.ARRIVAL_AIRPORT_IATA) Q2
    on Q1.DEPARTURE_AIRPORT_IATA = Q2.ARRIVAL_AIRPORT_IATA and
    Q1.FLIGHT_SERVICE_TYPE_IATA=Q2.FLIGHT_SERVICE_TYPE_IATA
group by Q1.FLIGHT_SERVICE_TYPE_IATA, Q1.DEPARTURE_AIRPORT_IATA
order by Q1.DEPARTURE_AIRPORT_IATA;
```

During the development of queries, an issue regarding cargo data was noticed. In TRDB, the structure is facilitated to store these data, but there is no data present in the associated columns. As of today, it is not possible to fulfil the wanted use case in terms of cargo. The query for cargo is based on the same as passengers and should work once the database is filled with the correct data. Even tho the data is not there, the execution time is included since the query runs fine, but produces no result.

3.1.2 Analysis

There are two wanted use cases for analysis on the data that is inside of TRDB. The first one is an analysis of credibility and reliability of different sources that produces or generates estimated time data to Avinor. These sources can be Eurocontrol, radar systems, airlines, etc. Sources might estimate time landing based on different variables and therefore essential to find the best one. The result from each source may vary depending on seasons, types of equipment, rain, snow, visibility, etc. Due to these variations, one analysis is not enough to determine which source is the best for estimation. It is, therefore, necessary to run such analysis frequently to have the best base guidance.

The second use case is to find out where and when a delay on a flight happened. A delay on one airplane can and probably will create a delay on future flights unless they can catch up on their planned schedule. The responsibility of updating the cause of the delay is up to the airlines themselves. Avinor has no influences on this and is not able to tell if the given cause is the right one. By tracking all of the flight by their registration number on a single day, it is possible to see which and what kind of delays they get for a flight. By doing this, Avinor might be able to find the exact reason for the delay and make improvements on the airports to fix specific issues. It would also be possible to see which airports have the most delays, and take specific actions based on this information. A

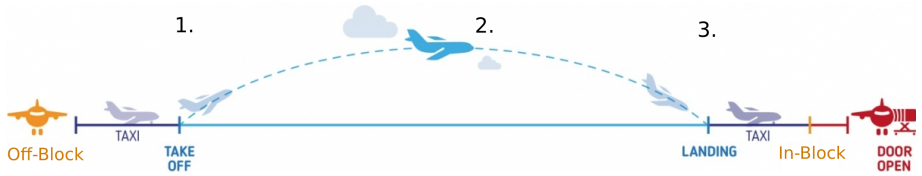


Figure 3.1: By checking take off time and flight time, the estimated landing time can be evaluated. Source: [2].

proper frequency for these queries would be once a day. This would make sure that there is sufficient data to do such an analysis.

One of the two examples can be seen below in the structured query:

```
select f.SOURCE_TIMESTAMP ,f.FLIGHT_ID, f.AIRCRAFT_REGISTRATION,
  f.DEPARTURE_AIRPORT_IATA, f.ARRIVAL_AIRPORT_IATA, f.SOBT,
  f.EOBT, f.AOBT, f.SIBT, f.EIBT, f.AIBT, d.DELAY_SEQ,
  d.DELAY_REASON, d.DELAY_DURATION
from FLIGHT_LEG_DATA_TAB f join FLIGHT_LEG_DEP_DELAY_TAB d
  on f.FLIGHT_LEG_DATA_ID = d.FLIGHT_LEG_DATA_ID
where trunc(FLIGHT_DEPARTURE_DATE) = trunc(sysdate)
  and f.LATEST_VERSION = 'Y'
order by f.AIRCRAFT_REGISTRATION, f.SOURCE_TIMESTAMP, d.DELAY_SEQ;
```

3.1.3 Real-time Analysis

Again, two different use cases would fit into the real-time analysis category. The first use case is to check what data is missing from flights that are soon to be departing from an airport. If Avinor detects that data is missing from an airline, appropriate actions can be taken. If they do not receive correct data, like for example passenger data, Avinor by default assumes that the flight is full and will demand tax fee from the airline based on that, whether or not the airplane is full. Warning airlines about missing data can save them a lot of money and Avinor can get better information about needed crew staffing.

The second case is to detect if flights are going to be delayed or not. By checking the actual take-off time(1.), see Figure 3.1, and calculating the travel time between airport A and airport B (2.), the estimated time for landing(3.) can be evaluated. If the estimated time for landing is far off, the likelihood for a delay is bigger then it would normally be. Detecting a delay faster would give Avinor to take appropriate actions ahead of time and make sure that the arriving airport can handle the delayed airplane.

It would not make sense to run this analysis on all of the data that is stored in TRDB. It is only flights that are soon to be departing from the airport and those who just took of, that needs to be check for deviations. The queries should be run with an interval of a few minutes. This would limit the number of flights that needs to be checked and account for data from third-party companies that send their data close to the take off time.

The queries that have been constructed for this use cases look like this:

```

select FLIGHT_ID, DEPARTURE_AIRPORT_IATA,
  ARRIVAL_AIRPORT_IATA, ATOT, ELDT, AIBT
```

```
from FLIGHT_LEG_DATA_TAB
where ATOT is not null
  and ELDT is not null
  and AIBT is null
  and LATEST_VERSION = 'Y'
  and trunc(FLIGHT_DEPARTURE_DATE) = trunc(sysdate);
```

3.1.4 Reporting

Central Office of Delay Analysis (CODA) is made by Eurocontrol and has a wide variety of different goals that they want to achieve when it comes to delays in the European airspace. Their main goal is to provide managers and partners of the ECAC Air Transport System with comprehensive information about the delay situation in Europe. This information is published and made available to anyone with interest in delay performance [42]. All of the information that is stored on Eurocontrol's internal databases and information received from airport operators in Europe are gathered and analyzed every month. The reports are useful for all partners to improve the punctuality for flights [43].

Avinor is one of many airport operators that sends their data to Eurocontrol. This data contains information about all delays for every flight in a given month. Avinor uses two different formats, one for Oslo Airport (IR390-APDF) and one for other airports (IR691).

The created query is based on the format of Oslo Airport (IR390-APDF):

```
select f.AIRCRAFT_REGISTRATION, f.AIRCRAFT_ICAO_TYPE, f.FLIGHT_ID,
  f.DEPARTURE_AIRPORT_ICAO, f.DEPARTURE_AIRPORT_IATA,
  f.ARRIVAL_AIRPORT_ICAO, f.ARRIVAL_AIRPORT_IATA, f.AOBT, f.ATOT,
  f.FLIGHTRULE, f.FLIGHT_SERVICE_TYPE_ICAO, f.SOBT,
  f.RUNWAYDEPARTURE, d.DELAY_CODE, d.DELAY_DURATION,
  f.FLIGHTLEGSTATUS, f.FLIGHT_SERVICE_TYPE_IATA, f.IFPLID
from FLIGHT_LEG_DATA_TAB f join FLIGHT_LEG_DEP_DELAY_TAB d
on f.FLIGHT_LEG_DATA_ID = d.FLIGHT_LEG_DATA_ID
where f.DEPARTURE_AIRPORT_IATA = 'OSL'
  and extract(Month from FLIGHT_DEPARTURE_DATE) = extract(Month from sysdate)
  and extract(Year from FLIGHT_DEPARTURE_DATE) = extract(Year from sysdate)
  and f.LATEST_VERSION = 'Y'
order by f.FLIGHT_LEG_DATA_ID desc, d.DELAY_SEQ asc;
```

3.1.5 Tax Calculation

As mentioned earlier, all flights in Norwegian airspace has to pay taxes. Several different variables define the amount of tax that should be paid. This data is intended to be stored in TRDB, and can replace an older existing system. The problem is that it is not possible to do tax calculation based on the data that is received and stored. Flight ID, registration number, operating airline and passenger number are a few examples of what is needed. This data is collected, stored and can be used as of today. The problem is that there is no information about the location or border crossing. As can be seen from Figure 2.3, there is a table that is called *FLIGHT_LEG_BCROSSING_TAB* and should contain the necessary data. However, this table is empty. This means that there is no way of exactly telling when

a flight enters the Norwegian airspace. Therefore, TRDB is not a viable solution for tax calculation at this point.

Eurocontrol sends out the necessary data through ETFMS Flight Data messages. These messages are distributed by broadcasting messages to everyone, and the receiver is responsible for filtering out required data [44]. Avinor has access to them, but have not prioritized them yet. Once this is done, tax calculation could be done.

3.1.6 Passenger prediction

Beontra is a planning and forecasting system that connects to data sources like TRDB and passenger tracking systems (e.g. from Bluetooth or WiFi) to merge these sources into a reliable base for a continuous, automated forecast of passenger streams in a terminal [45]. As of today, Beontra runs as a cloud solution and receives data for the major airports located in Oslo, Stavanger, Bergen, and Trondheim. All scheduled and chartered flights, both arriving and departing flight are sent to Beontra two times every week.

The created query is based on the report that is generated and sent to Beontra as of today:

```
1  select f.OPERATING_AIRLINE_IATA, f.OPERATING_AIRLINE_ICAO,
2    f.FLIGHT_ID, f.FLIGHT_SERVICE_TYPE_IATA, f.ATOT, f.SOBT,
3    f.ARRIVAL_AIRPORT_IATA, f.ARRIVAL_AIRPORT_ICAO, f.TERMINAL_DEP,
4    f.TERMINAL_ARR, g.GATE, f.AIRCRAFT_IATA_TYPE,
5    f.AIRCRAFT_ICAO_TYPE, f.AIRCRAFT_SEATING_CAPACITY,
6    f.AIRCRAFT_MTOW, f.PAX_ADULTS_ON_BOARD, f.PAX_CHILD_ON_BOARD,
7    f.PAX_INFANT_ON_BOARD, f.PAX_TRANSIT, f.DEPARTURE_AIRPORT_IATA
8  from FLIGHT_LEG_DATA_TAB f join FLIGHT_LEG_GATE_TAB g
9    on f.FLIGHT_LEG_DATA_ID = g.FLIGHT_LEG_DATA_ID
10 where trunc(f.FLIGHT_DEPARTURE_DATE) between
11   trunc(sysdate-90) and trunc(sysdate)
12   and LATEST_VERSION = 'Y';
```

3.2 Baseline result

The problem that Avinor reports with TRDB is the execution time of queries. On average, the response time is somewhere near 60 seconds or above. Therefore each query has been measured by its execution time. Each query has been executed five times to make up for any variation during run time. The total average for each query execution time is documented in Figure 3.2.

The results from each query were collected from the database system itself. There is an internal table in the existing RDBMS that keeps statistics over each query that is executed. In this experiment, each query got measured by;

```
CPU_TIME, USER_IO_WAIT_TIME, APPLICATION_WAIT_TIME,
CONCURRENCY_WAIT_TIME, CLUSTER_WAIT_TIME
and PLSQL_EXEC_TIME.
```

Based on the result that can be seen in Figure 3.2, it is possible to see that only *CPU_TIME* and *USER_IO_WAIT_TIME* has a significant impact on the execution time. All other measurements returned zero as its value and have been left out based on this.

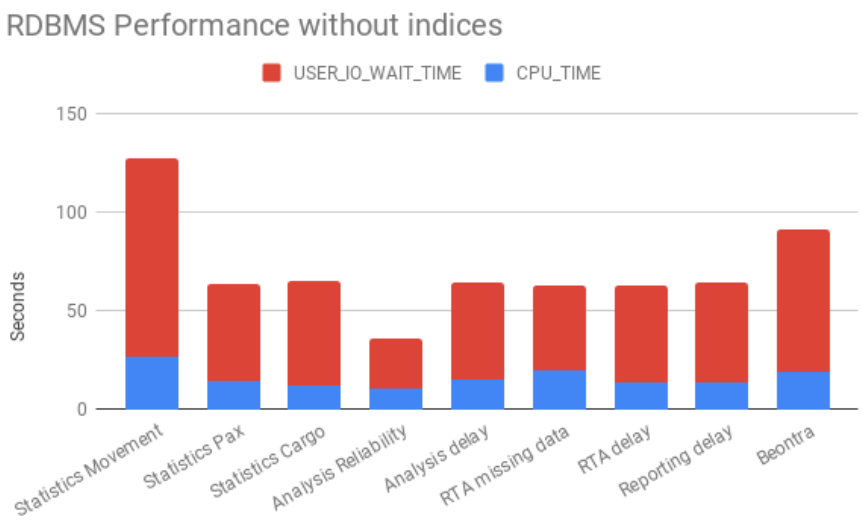


Figure 3.2: Average execution time with no indexes.

3.2.1 Discussion on Baseline result

It is essential to point out that these use cases that are mentioned above are just a subpart of all cases. It has been a focus to get used cases from different categories, to get a variety in need of data. The limitation is due to the time and resources of this thesis. However, all of the queries that have been made will be used further in this thesis to test improvement that are implemented on the database. The result from each implementation will be compared to each other. This may not be a final version of what the queries would look like if Avinor where to use them in their systems.

Much data are missing from the database. There are 218 columns in the main table, but over 110 of these are always empty. This causes problems for use cases and has been mentioned earlier in the chapter. In order to make TRDB a viable solution for the defined use cases, work needs to be done in order to receive and store all data. Most of the data is already available from different messages that are sent internally or from Eurocontrol, but these need to be processed further to get the necessary data that is missing.

The queries that have been talked about and needed for the use cases are simple queries, with maximum one join in each query. Even tho the queries are simple, the execution time is enormous. In the next section, a look at how indexes can impact performance will be presented.

3.3 Indexes

An index for a table is a data organization that enables certain queries to access one or more records of that table fast. Proper tuning of indexes is therefore essential to high-

performance [46]. However, the RDBMS offers a wide variety of index structures, each suitable for different use cases. This section will look at different structures to determine which one is the most suited in the use cases that were talked about in Section 3.1.

3.3.1 B+-Tree Index

B+-tree indexes are known as the standard or general index in the RDBMS. B+-tree is short for balancing tree and tries to minimize the amount of time that is spent on searching for data. The lowest level blocks in the tree, called leaf nodes, contains every indexed key and a row id that points to the row it is indexing [47]. The blocks above the leaf nodes are used to navigate the structure. Items in the leaf nodes are ordered and linked, meaning that a range scan of values is straightforward. By using a B+-tree, it is only necessary to find the first item in the leaf nodes that satisfies the search criteria and iterate horizontally through the links of leaf nodes and stop whenever an item does not satisfy the search criteria any more.

Because a B+-tree is versatile, it can be used in many cases. In these cases, it could be viable to have B+-tree indexes on arriving and departing airports or number specific columns like passenger numbers. Only index for arriving and departing airport has been made.

3.3.2 Bitmap Index

A bitmap index is usually used on relations that contain a large number of rows. Since TRDB has a large number of rows, bitmap might be a viable option. Typically, a bitmap index is created for those columns that contain a fairly small number of unique values [48]. Based on the queries, there are no columns that fit this description. The column for departing or arriving airports would be the closest in consideration. As of now, there are 612 different airports, which would not be looked upon as a small number of unique values. However, if there would be a column in the future that only included Avinor's 44 airports, this would have been a more viable option as bitmap index. For this thesis, bitmap index was never used.

3.3.3 Function-Based Index

With function-based indexes, it is possible to run functions on data before the data is being indexed. Developers themselves can create functions or use functions that are available in the RDBMS. They are easy to implement, provide immediate value and speed up existing applications without changing the logic or queries [47].

Many of the queries that are described in Section 3.1 uses data from date fields. A problem with indexing date fields is that they contain the time element. If they were to contain the time element, there would be 86.400 different values for each day. This means that it is very likely that a new index value would be made, each time a new row is inserted. This would result in a large index table and not improve performance at all for the database.

By using function-based indexes, this can be avoided to a certain extent. *TRUNC()* is a function that removes or truncates the timestamp in the date field. This will result in only one index value for each day compared to the worst case, which is 86.400, as mentioned

above. Most queries are not dependent on the timestamp since they want hits in a single day or multiple days. Because of this enormous difference, date fields that are used will be indexed by function-based indexes. *EXTRACT()* is another example of a function that is useful on date fields. The function allows the developer to extract months or year of the date field. Using this index on queries that will be run each month, improves the performance significantly and can be seen in 3.3.6.

However, there is a drawback using function-based indexes. For each insert or update, the index value has to be computed. Function-based indexes are not recommended for write-intensive databases due to this problem. TRDB will query data frequently, thus making it more suitable for function-based indexes. According to T.Kyte, the pros of function-based indexes heavily outweigh any of the cons when querying data frequently [47].

3.3.4 Application Domain Index

Application domain indexes allows the developer to define a new index structure on their own. Examples of where this could be used would be; documents, spatial data, images or video clips [49]. The developer of such an index would have to create the code that extracts the needed data from the source. There is no need for this in TRDB, and will therefore not be used as an index on any of the columns.

3.3.5 Reduce Amount of indexes

In some instances, adding too many indexes to columns or tables, will create worse performance then already obtain, which an example can be seen in the blog written by Chris Saxon [50]. It is therefore important to never have more indexes then what is needed, but have the correct index on the data that is being used. In order to avoid unnecessary indexes, a few of the queries was rewritten to use the same indexes that other queries used.

Since indexes are stored in physical tables on disk, it also reduces the amount of space that is needed to store all of the indexes.

It is, however, essential to point out that these indexes that were made were to improve the queries that are used for the baseline in this thesis. More indexes would need to be added for production setup at Avinor, but those indexes depend on frequently queries and usage of them. The indexes in this thesis are mainly to show the performance gap between an indexed database and a database without.

3.3.6 Results After Indexing Data

By simply adding a few indexes on the table, the execution time has been reduced significantly. The queries that had no joins, no range scan and no aggregation has been reduced down to less than a second, which used to be around 60seconds without indexes. Queries that need data for a month or more has a higher execution time due to the amount of data that is required, which can be seen by USER_IO_WAIT_TIME in Figure 3.3.

The result will be discussed in further depth in Chapter 5.

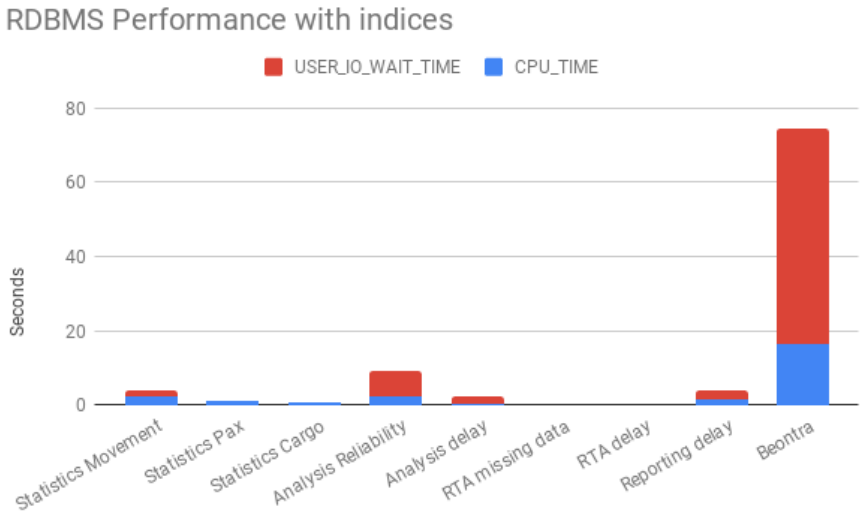


Figure 3.3: Execution time of queries with indexes.

Chapter 4

NoSQL Technology

This chapter will specify the reasoning for picking a NoSQL system and the alternatives, how it was structured based on the data, and how it performs. The result from the queries will be presented at the end of this chapter.

4.1 Choice of NoSQL Database

The NoSQL system that is used for this thesis is MongoDB[51]. The reasoning of choice will be presented in the subsections that are listed below. Other alternatives will be discussed at the end, and the reasoning for not picking them is presented in order to answer research question number 3.

4.1.1 Mapping between RDBMS & MongoDB

When talking about RDBMS and MongoDB, various terms are being used to describe the same concept. In Table 4.1, their similar terms have been paired with each other. The terms for MongoDB will be used forward when describing the structure and how the process was done. It is essential to know the difference between them and their existence in order to understand the next couple of sections.

4.1.2 Document-Based Database

MongoDB is a NoSQL system and structure data into documents. Each document in the database can be compared to an entry or row in an RDBMS, and usually contains many key/value pairs or key-array pairs. Data is fetched by looking for the correct key in documents and return it. Documents are stored in a BSON format, similar to a JSON, a format most developers will be familiar to. An example of such a structure can be seen in Figure 4.1. Looking at the data structure from Appendix II, it can be seen that TRDB gets the data in an XML format. XML can be compared to JSON, even tho XML has a few unique features. According to Srivastava, Goyal, and Kumar, MongoDB and its

SQL Terms	MongoDB Terms
Database	Database
Table	Collection
Row	Document
Column	Field
Index	Index
Aggregation (e.g. group by)	Aggregation pipeline

Table 4.1: The terminology between RDBMS and MongoDB is not identical. The equivalent terms have been paired together.

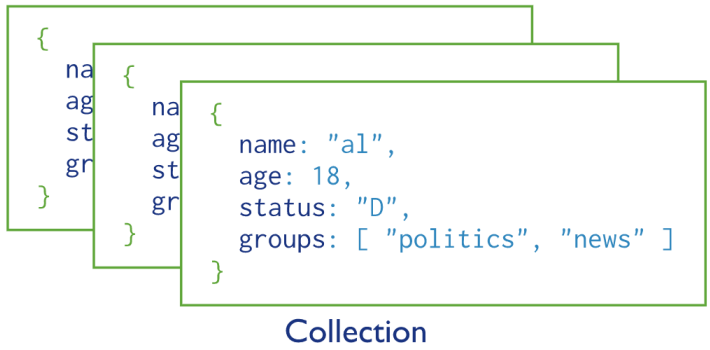


Figure 4.1: A document-based databases will consist of a collection of documents, with key-values stored inside each document. Each document is similar to a row in RDBMS. Source: [3]

document-based approach is best suited as an alternative to web application using RDBMS [52]. Based on the existing structure of data and the similarity to an RDBMS, the decision was taken to go with a document-based approach in this thesis. In addition, there are many XML to JSON converts, which could be implemented to lessen the structural changes that are required to go from an RDBMS to a NoSQL system. The three following subsection describes the main reason for why MongoDB was chosen as the document-based database in this thesis.

4.1.3 Popular NoSQL Database

The development of MongoDB started in 2007 and has gained much traction since then. A master thesis written by Kamil Kolonko in 2018 [53], found out that MongoDB is the most popular non-relational database management system. 310 queries were executed in the Google search engine, and the result from these was aggregated to find the total number of results. MongoDB had the highest number of returned result for a NoSQL database with a score of 167.000, compared to Oracle with the highest score of 9.430.000 for an RDBMS.

DB-Engine is a website that ranks all database system[54] based on several different metrics. As of April 2019, MongoDB was the fifth most popular database system, the most popular document-based storage and the most popular system for NoSQL database.

4.1.4 Enterprise Support

MongoDB offers extensive support to enterprises if wanted. According to their website, their global team is available all the time with diagnosing, detecting, and potential troubleshooting issues before they turn into problems. A MongoDB Named Technical Support Engineer (NTSE) will work closely with the enterprise, to provide dedicated support tailored to their unique technology and operational needs [37]. The options for support are many, but the price of this is unknown. The enterprise itself can choose these options if it is necessary.

It is essential for Avinor to have the option of enterprise support. Transportation and society in Norway are heavily dependent on the solution and work of Avinor. If their systems go down, it could have major consequences such as delays and at worst cancellations, and cause chaos for travellers. By having enterprise support the risk of such a failure is reduced since engineers are always available and ready to help.

In addition to having enterprise support, many well-known enterprises have chosen MongoDB as a part of their software development. eBay, Nokia, Adobe, Bosch, Google, Cisco, SAP, Facebook and many more are examples of huge enterprises that uses MongoDB [55]. This shows that MongoDB has received the trust, strength, and versatility of companies and is a competitive alternative to other RDBMS and NoSQL systems.

4.1.5 Features

MongoDB has many key features to offer. Master-slave replication and flexible schema is a few examples and was talked about in Chapter 2. In addition to these, MongoDB offers secondary indexes, a wide range of supported programming languages, immediately consistent if needed, concurrency, durability and in-memory capabilities[56]. In June 2018, support for multi-document transactions with snapshot isolation was added in MongoDB 4.0. This makes MongoDB fully ACID compliant, along with a few other NoSQL databases like RavenDB[57] and ArangoDB[58].

These features make it possible for Avinor to customize their database to their need. Immediately consistent can be used, but will reduce the performance of queries. This is because all nodes need to be updated before the transaction is committed and stored, and the network latency of confirmation.

In addition to these features, MongoDB also offers replication and auto-sharding. A closer look at these and how they can impact operations at Avinor will be presented.

Replication

MongoDB maintains multiple copies of the data in secondary nodes using replication. There are three main benefits to this. The first one is that there is always a backup node(secondary node) that can replace the primary node. If the primary node, see Figure 4.2, shuts off, the secondary nodes will notice that the primary nodes are not available

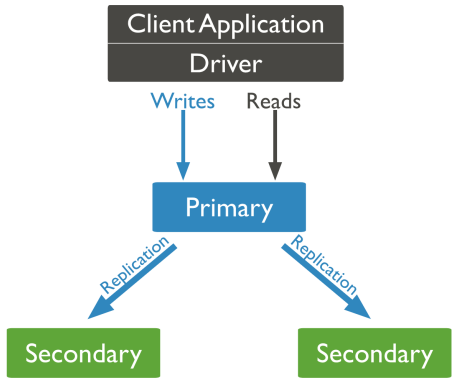


Figure 4.2: Replication of MongoDB. Source: [4]

and they will have an election to decide which of them will be the new primary node. There will, unfortunately, be a loss of data during the time from shut off. The election of a new primary is an automatic process and require no manual input. This ensures that the availability of service is as good as it can get and sufficient redundancy.

The second reason for having replication of data is to increase capacity in terms of reading operations. Not all queries need the newest data, and this can be taken advantage of. Secondary nodes will not receive updates immediately after committing in the client application. Reporting application is typically applications that accept data to be slightly out of date. These applications can read their data from the secondary nodes, thus reducing the workload for the primary node. An application that would need data in real time would have to use the primary in order to get the newest updates at any given time. All writes would still be directed to the primary node.

The third benefit is that nodes can be maintained and repaired without issues. Hardware and software can be upgraded or replaced without the database going down. According to the MongoDB Architecture [4], these operations can stand for as much as 1/3 of all downtime in traditional systems.

For Avinor, this means that they do not need to think much about backup and replication of data, they can handle a higher workload, and repairs and upgrades can be done to any node whenever wanted.

Auto-scaling

MongoDB has automatic horizontal scaling of the database built-in. The technique used is called *sharding*. Sharding distributes data across physical partitions called shards [4]. The main reason for this is to avoid bottlenecks that would occur on single servers such as RAM or disk I/O. The distribution of data is done automatically by MongoDB whether the data grows or the size of the cluster increases. Horizontal scaling of MongoDB is displayed in Figure 4.3 and shows that the capacity can be increased.

Since the process is automated, developers do not need to consider this when making application code and database administrators do not need to deploy clustering software on

Figure 4.3: Auto-scaling of cluster in MongoDB. Source: [4]

the computers.

The data can be sharded in a few different ways.

- It can be sharded according to a shard key value, making it optimized for range-based queries, but the workload is reduced to one or a few nodes.
- It can be sharded based on an MD5 hash of the shard key, such that data is guaranteed a uniform distribution across shards making it suitable for balanced workload, but less optimal for range-based queries.
- It can be sharded based on user-specific configuration such as specific data centre or storage types, making memory the storage for most used data and HDD for stale data.

Since data is sharded across multiple shards, MongoDB depends on the *Query Router*, see Figure 4.4, to dispatch the application query to the correct shard. If the query is based on the shard key, it is only the shard that contains the data that receives the query. This means that for key-value and range-based queries with shard key, the query router will only dispatch the query to those shards that contain the data. If the query does not contain the shard key, the query will be broadcasted to all shards, and the query router will aggregate the result from incoming shards. The query router, therefore, needs to know which shards contains which data and be able to contact all of the nodes in the network.

For Avinor, this means that a straightforward scale-up of the system can be done by adding an addition node to the network, and MongoDB will handle data distribution and workload accordingly.

4.1.6 Other NoSQL categories

There are many NoSQL databases and categories that could have been feasible for this thesis, and each of them will now be looked at in order to determine why these categories was not suited or why they where not chosen.

Key-values databases have their data stored in the format of a key, and values that belong to the key. It is only possible to search for keys and not on values. An example of the structure can be seen in Figure 4.5. The use case of a key-value database could be session management of web applications, massive online gaming session or shopping carts for online users of a web application [59], where data is never queried by anything other than a primary key. Examples of existing databases are; Amazon DynamoDB[60], Redis[61], RocksDB[62]. Multiple items are searched for in each query in TRDB, and a simple key would not be sufficient enough for the use cases. Therefore, it was concluded

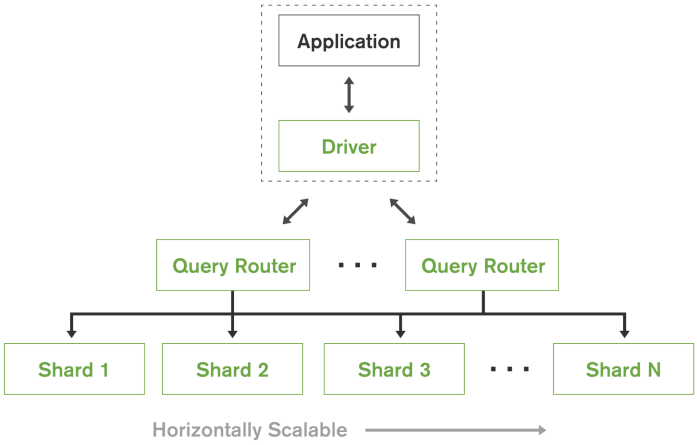


Figure 4.4: Connection between application, query router and shards in MongoDB. Source: [4]

that a key-value databases would not be an viable option in the case of TRDB and its use cases.

Graph database have their data stored as a graph. Each node in the graph is an entity or object, while edges is a relationship between the entities. An example of such structure can be seen in Figure 4.6. Red, green and blue nodes represents different entities, and the edges represent the relationship between them. Examples of existing databases are; Neo4j[63] and OrientDB[64]. According to Robin Hecht and Stefan Jablonski, use cases for graph databases are location-based services, knowledge representation and pathfinding problems raised in navigation systems, recommendation systems and all other use cases which involves complex relationships[65]. In the case of Avinor, an airport might be represented as a node in the graph and each aircraft movement as an edge between two nodes. According to Srivastava, Goyal, and Kumar, graph databases are best suited for traversing and search application. The relationship between data is more important rather than the data itself [52]. In addition, they are efficient for graph algorithms such as shortest path link between information. The data itself in TRDB is more important than the relationship between them in terms of usefulness. Based on these statements that are presented in the papers, a graph database would not be the best option for Avinor based on the use cases.

Column Family databases, also known as extensible records, is similar to an RDBMS. Instead of having data in row format, the data is structured in column format. RDBMS are designed to return entire rows of data when fetching efficiently. When using a SATA hard drive, rows are being read when fetched. The seek time of hard drive is usually the bottleneck in computers; something column family databases want to reduce. Conceptually, each column has its index, such that the database knows where to look for the data on a hard drive. By storing data in columns, all data in a column is stored sequentially on a hard drive. This reduces the seek time since data is stored sequentially. A good example between the difference from an RDBMS and a column family database can be seen in Figure 4.7. A few examples of databases using column familie are; BigTable[66], Hbase[67] and Cassandra[68]. The advantage is that the read time is reduced when only reading a

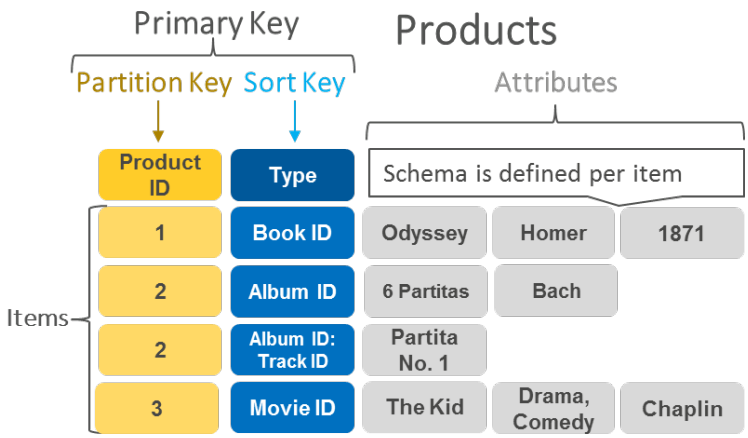


Figure 4.5: Example of Amazon DynamoDB with its primary key and attributes/values. Source: [5]

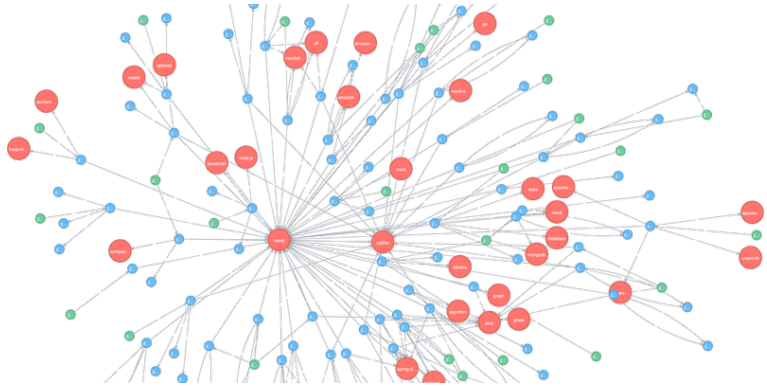


Figure 4.6: An example of a graph database structure. Colored nodes represents different entities and edges represents relationship between them. Source: [6]

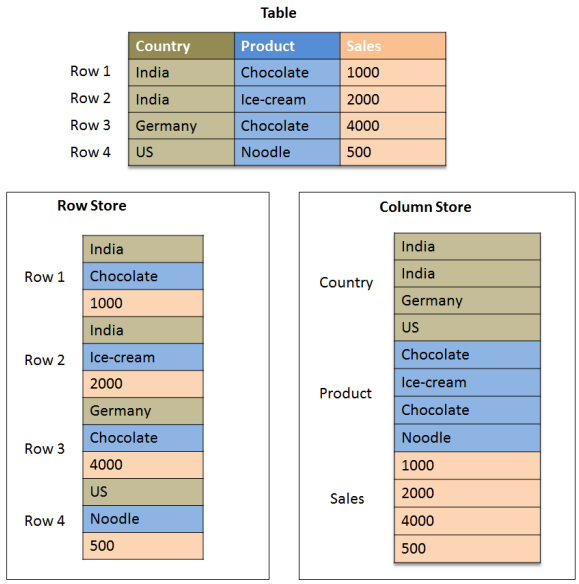


Figure 4.7: On the left side the structure of an RDBMS is presented. On the right side the structure of an Column Family database is presented. Source: [7]

few columns and better compression since similar data is stored next to each other on disk. The disadvantages are that the insert time of new data is increased since all columns need to be updated. Column Family databases are more suited for Online Analytical Processing(OLAP) operations, meaning aggregation over colossal data sets [69]. This is not the use case in TRDB, and therefore, column family databases is not suitable.

4.2 Data Structure

Since MongoDB requires data in JSON/BSON format, a few modifications had to be done on the data structure. The only way to retrieve all of the necessary data was through the existing RDBMS and its UI client. All tables from Figure 2.3 was exported as separate CSV files. By using Python[70] and a library called Pandas[71], each flight was merged with its corresponding data, into documents. All of the child tables in Figure 2.3 was structured into nested values. The process of converting rows to JSON documents was time-consuming and required several hours to finished. In total, the program used 41 hours to convert all of the data to JSON. The reason for this is the limitation in memory on the computer. With a total of 12GB, the entire data set was not able to fit into memory. The process was therefore broken into smaller parts, to make sure that it would succeed. It is also important to point out that the program that was made was not threaded, thus reducing the capability of the CPU. This would have sped up the process since the CPU was the bottlenecks in certain periods.

In Figure 4.8, a flowchart of the explained solution is shown. An alternative would be

to have a converter from XML to JSON/BSON and insert that directly to MongoDB. That would reduce the overhead of steps and chances of error would be reduced significantly. This solution is represented with dotted lines in the flowchart, Figure 4.8. This alternative was not chosen since data for all *FlightLegStateMessage* was not available.

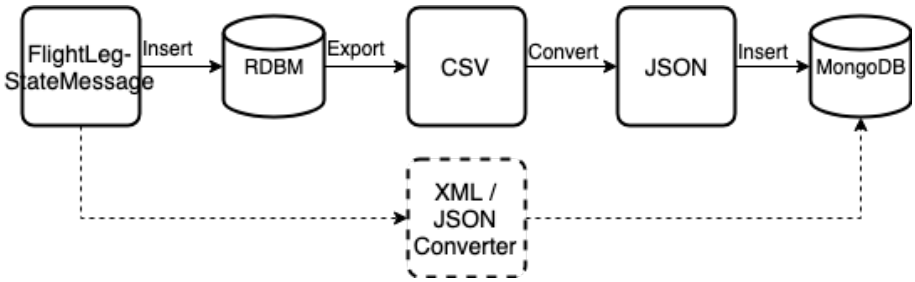


Figure 4.8: Flowchart from RDBMS to NoSQL.

MongoDB offers a variety of documents related to conversion from RDBMS to NoSQL in terms of specification, transformation and maintenance. In order to make sure that the process for schema translation was correct, the official document from RDBMS to MongoDB Migration Guide was used [72]. An example of a subset of the structure of a document can be seen below:

```
{
    "FLIGHT_LEG_DATA_ID": 45,
    "ARRIVAL_AIRPORT_IATA": "SVG",
    "DEPARTURE_AIRPORT_IATA": "OSL",
    "FLIGHT_DEPARTURE_DATE": 1456786500000,
    "LATEST_VERSION": "Y",
    "SIBT": 1451779200000,
    "AOBT": null,
    "ATOT": null,
    "PAXSEATEDONBOARD": 0,
    "DELAY" : {
      "DELAY_SEQ" : 1
      "DELAY_REASON" : "Pax missing"
      "DELAY_DURATION" : 10
    }
}
```

As can be seen in the example above, there are a few values that are *null*. When creating a JSON format, all keys with nothing/empty as its value will be assigned a *null* value. In fact, from the RDBMS, there are over 110 columns that are always empty. All these *null* values are not needed in the data set and use additional storage. Before the data was imported to MongoDB, a bash command was created in order to remove all unnecessary values. The bash command used a series of Linux based commands and was piped together to accomplish all of the formatting and creation of new files. The command can be seen in Figure 4.9. Before using this command in Linux, the total size of the documents was 62,1GB. After removal of *null* values, the total size was reduced to 51,9GB, which is a reduction of 10,2GB. Once the data was imported to MongoDB, the storage size was only 32,4GB. The reason for variation is due to compression of data in MongoDB. The storage

is higher than in RDBMS because compression is different between them and that keys are stored for each value.

```
bortne@Bortne:~$ for i in resultTest*; do cat $i | json_reformat |
sed '/: null/d' > "cleaned-"$(echo $i |sed -e s/[^0-9]//g)".json";
done_
```

Figure 4.9: Bash command used to format documents before import.

In addition to removing *null* values, the converting from RDBMS to JSON format removed many of the field types. When importing JSON documents to MongoDB, MongoDB will try to guess or assume the data type based on its value. This method works most of the times. Unfortunately, date formats did not work. The RDBMS and MongoDB interpret date format in two different ways. To solve this, all date fields were transformed into UNIX timestamp when going from CSV to JSON. An example can be seen above with *FLIGHT_DEPARTURE_DATE* and *SIBT*. When imported, MongoDB would assume that the value was a *int* and not a *date*. A script was made to change all date values from UNIX timestamp to MongoDB date in order to have the correct date format.

In order to make sure that both of the databases contained the same data set, the number of rows and documents was check, in addition to unique numbers of rows and documents.

4.3 MongoDB Performance

In order to compare the RDBMS with MongoDB, the performance needs to be measured. The foundation for the performance tests will be as similar to the one in RDBMS, but with a few variations to facilitate for MongoDB. The main focus will be to look at reading performance without and with indexes.

4.3.1 Use Cases

The use cases which are defined to measure read performance are the same as described in Chapter 3.1. The syntax and structure of queries are very different from what is used in an RDBMS, but the returning result is the same or as similar as it gets. All of the functions that are used in the queries below can be found in the official manual for MongoDB 4.0 [73]. Not all queries are included in this chapter, but one from each category. The baseline result can be seen in Chapter 4.3.2.

All of the queries were made with PyMongo. PyMongo [74] is a distribution that contains all of the tools that are necessary to work with MongoDB from Python. Python was chosen as the preferred query language due to their popularity in the developer community [75] and its open source.

Statistics

When talking about statistics and aggregation of data, it is important to understand the underlying framework for how MongoDB handles these kinds of operations. MongoDB

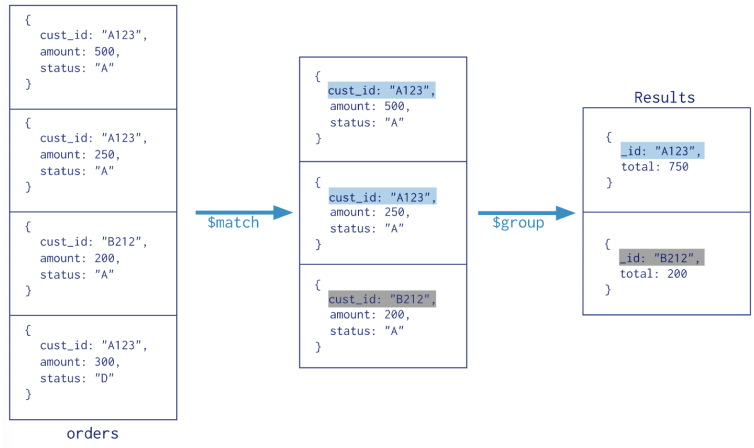


Figure 4.10: Example of the MongoDB pipeline. Source: [8]

uses aggregation pipeline, which is a framework for data aggregation that transforms documents into aggregated results through separate stages in the pipeline. An example of this can be seen in Figure 4.10.

As can be seen from Figure 4.10, the collection of documents are reduced based on criteria or filters, and the documents left are aggregated into a result document. This approach is used when creating queries for statistics use cases.

There is, however, some limitation to the approach since it uses the pipeline aggregation framework. To find the flight service types from airports in RDBMS, ref 3.1.1, a join of two separate queries was used to receive the wanted result. This would not be able to achieve since MongoDB does not support joins and it would need to be joined in the application code in order to get the same result. The other statistics use cases worked fine with the pipeline framework.

A subpart of the query for statistics on flight service types can be seen below:

```
statMovementDep = collection.aggregate([
  {'$match' : {
    'FLIGHT_DEPARTURE_DATE' : {
        '$gte'   : startDate,
        '$lt'    :  endDate},
    'LATEST_VERSION' : 'Y',
    'AOBT' : {'$exists' : True}}
    },
    {'$group' :
        {'_id' : {'FLIGHT_SERVICE_TYPE_IATA' : '$FLIGHT_SERVICE_TYPE_IATA',
                  'DEPARTURE_AIRPORT_IATA' : '$DEPARTURE_AIRPORT_IATA'
                  },
    'count' :
        {'$sum' : 1}}
    },
    {'$sort' : {'DEPARTURE_AIRPORT_IATA' : 1}}
])
```

Analysis

The analysis query constructed is a range-query that spans across a day in order to see when and where an airplane got delayed. Lines 1-3 filter out wanted documents, similar to *where* in SQL. Lines 4-8 is there to specify what key-values are wanted to return and would be equivalent to a *select* in SQL. The last two lines sort all of the documents based on aircraft registration, time stamp and delay sequence. This makes it easier to find all of the delays for a specific aircraft for a given day.

```
1  analysisDelay = collection.find({'LATEST_VERSION': 'Y',
2      'FLIGHT_DEPARTURE_DATE' : {'$lt': datetime(2016, 4, 27),
3      '$gte': datetime(2016, 1, 26)}},
4      {'SOURCE_TIMESTAMP' : 1, 'FLIGHT_ID' : 1, 'AIBT' : 1,
5      'AIRCRAFT_REGISTRATION' : 1, 'DEPARTURE_AIRPORT_IATA' : 1,
6      'ARRIVAL_AIRPORT_IATA' : 1, 'SOBT' : 1, 'EOBT' : 1,
7      'AOBT' : 1, 'SIBT' : 1, 'EIBT' : 1, 'DELAY:DELAY_SEQ' : 1,
8      'DELAY:DELAY_REASON' : 1, 'DELAY:DELAY_DURATION' : 1, '_id': 0})
9      .sort([('AIRCRAFT_REGISTRATION', 1),
10     ('SOURCE_TIMESTAMP', 1), ('DELAY:DELAY_SEQ', 1)])
```

Real-time Analysis

The real-time analysis query checks which aircraft has departed from the airport, but not reached their destination yet. If the destination is not reached, it can calculate the estimated travel time between airports and the take-off time to predict if the aircraft will be on time or not. Only needed key-values from each document is extracted.

```
1  RTADelay = collection.find({'LATEST_VERSION': 'Y',
2      'FLIGHT_DEPARTURE_DATE' : {'$lt': datetime(2016, 1, 27),
3      '$gte': datetime(2016, 1, 26)},
4      'ATOT' : {'$ne' : None}, 'ELDT' : {'$ne' : None},
5      '$or' : [
6          {'AIBT' : None},
7          {'AIBT' : {'$exists' : False}}
8      ]},
9      {'FLIGHT_ID' : 1, 'DEPARTURE_AIRPORT_IATA' : 1,
10     'ARRIVAL_AIRPORT_IATA' : 1, 'ATOT' : 1, 'ELDT' : 1,
11     'AIBT' : 1, '_id': 0})
```

Reporting

The reporting query will be used for the analysis of delay by Eurocontrol. The query is a range-based query that returns all movement within a given time frame. In this case, the default time frame is one month. Only needed key-values from each document is extracted.

```
1  reportingDelay = collection.find(
2      {'LATEST_VERSION': 'Y', 'FLIGHT_DEPARTURE_DATE' :
3      {'$lt': datetime(2016, 3, 26, 0, 0),
4      '$gte': datetime(2016, 2, 26, 0, 0)},
5      'DEPARTURE_AIRPORT_IATA' : 'OSL'},
6      {'AIRCRAFT_REGISTRATION' : 1, 'AIRCRAFT_ICAO_TYPE' : 1,
7      'FLIGHT_ID' : 1, 'DEPARTURE_AIRPORT_ICAO' : 1,
```

```
8       'DEPARTURE_AIRPORT_IATA' : 1, 'ARRIVAL_AIRPORT_ICAO' : 1,
9       'ARRIVAL_AIRPORT_IATA' : 1, 'AOBT' : 1, 'ATOT' : 1,
10      'FLIGHTRULE' : 1, 'FLIGHT_SERVICE_TYPE_ICAO' : 1,
11      'SOBT' : 1, 'RUNWAYDEPARTURE' : 1, 'DELAY:DELAY_CODE' : 1,
12      'DELAY:DELAY_DURATION' : 1, 'FLIGHTLEGSTATUS' : 1,
13      'FLIGHT_SERVICE_TYPE_IATA' : 1, 'IFPLID' : 1, '_id' : 0})
```

Tax Calculation

As stated in Section 3.1.5, the data quality is not good enough to do tax calculation the way intended from Avinor. Since it is not possible to achieve the wanted result in RDBMS, it is not possible to achieve it in MongoDB either. It is, therefore, no queries made for this use case since it is not possible to do as of now.

Passenger Prediction

Quite similar to subsection about reporting, passenger prediction is also a range-based query that returns all movement within a given time frame. The main difference here is the time frame. In this case, the default time frame would be three months or 90 days. Also here the only needed key-values is extracted from the returned documents as can be seen in lines 5-14.

```
1  beontra = collection.find(
2       {'LATEST_VERSION': 'Y',
3       'FLIGHT_DEPARTURE_DATE' : {'$lt': datetime(2016, 4, 26),
4       '$gte': datetime(2016, 1, 26)}},
5       {'Flight_Departure_Date' : 1, 'OPERATING_AIRLINE_IATA' : 1,
6       'OPERATING_AIRLINE_ICAO' : 1, 'FLIGHT_ID' : 1, 'ATOT' : 1,
7       'FLIGHT_SERVICE_TYPE_IATA' : 1, 'SOBT' : 1,
8       'ARRIVAL_AIRPORT_IATA' : 1, 'ARRIVAL_AIRPORT_ICAO' : 1,
9       'TERMINAL_DEP' : 1, 'TERMINAL_ARR' : 1, 'GATE:GATE' : 1,
10      'AIRCRAFT_IATA_TYPE' : 1, 'AIRCRAFT_ICAO_TYPE' : 1,
11      'AIRCRAFT_SEATING_CAPACITY' : 1, 'AIRCRAFT_MTOW' : 1,
12      'PAX_ADULTS_ON_BOARD' : 1, 'PAX_CHILD_ON_BOARD' : 1,
13      'PAX_INFANT_ON_BOARD' : 1,'PAX_TRANSIT' : 1,
14      'DEPARTURE_AIRPORT_IATA' : 1, '_id' : 0})
```

4.3.2 Baseline Result

To determine the measurement of each query for MongoDB, the built-in functionality of the explain plan has been used. *Explain()* returns a document with results from the query optimizer in MongoDB, that includes the winning plan for the operation under evaluation along with its execution statistics [76]. This approach was chosen to eliminate any third party software that might have influenced the result and to make sure that the result came straight from the source.

Explain.executionStats.executionTimeMillis is a subpart of the returned document after calling *explain()* that measure the execution time. The time is measured in milliseconds

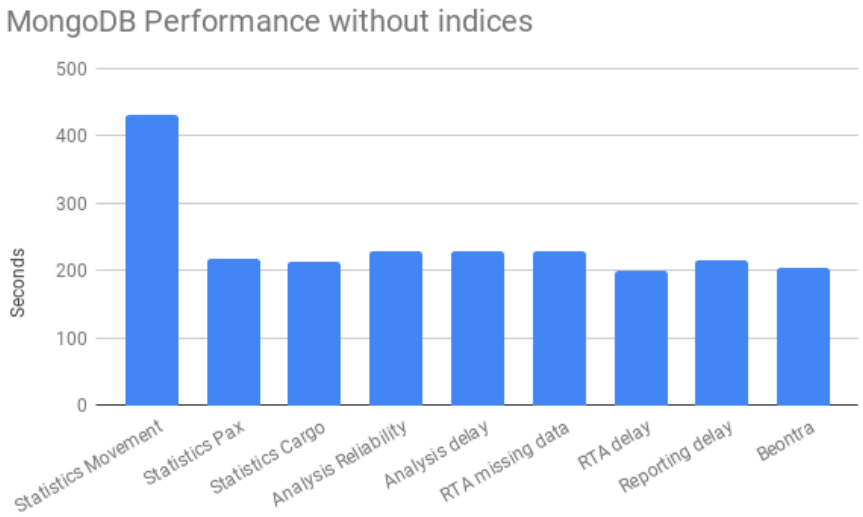


Figure 4.11: MongoDB Performance without indexes.

and is a combination of query plan selection and query execution. Based on the documentation, it is not possible to only get one of them or to separate the result. It is, therefore, unknown how much time is being spent on each part.

The baseline result from each query in MongoDB can be seen in Figure 4.11.

4.3.3 Indexes

Indexes in MongoDB is similar to any other indexes in a database system. Indexes limits the number of documents that must be inspected in order to return a result. Without these, MongoDB must scan every document in the entire collection, which is time-consuming and will affect the read performance. This impact can be seen from the baseline result. As with the RDBMS, there is a variety in index types that will be looked at.

Single Field Index

Single field index is the simplest index that MongoDB creates. This is only based on one field in a document and can be sorted ascending or descending. The sort of indexes on single field index does not matter since MongoDB traverses the indexes in either direction [77]. It was found out that this type of index was sufficient enough for all use cases and indexes was added on the same columns as with the RDBMS.

Compound Index

Compound indexes create indexes based on multiple values defined by the developers. This means that two fields can be indexed together based on their values. It is also possible

to determine the order of sort for each field. Since the use cases do not require such an operation, this index is not needed.

Multikey Index

Multikey indexes are usually used for a value that contains an array. MongoDB can automatically determine whether a value is an array or not, thus creating the index that is most suited and no more specification is needed. Since there are not array values in the data set of TRDB, this index will not be needed for this experiment, nor in the future, unless further changes are made.

Geospatial Index

MongoDB offers geospatial index on geospatial data. It is not an option to use for TRDB since there is no geospatial data there, but it could be used. TRDB is intended to store location data from flights, but there is no data available. However, if there was any location data, which might be fixed in the future, a geospatial index might be wanted. This could be used for visualization of flight movement and placement or find out if any flights have taken any routes they were not supposed to take.

Text Index & Hashed Index

One of the available options is text index and hashed index. Text index supports searching for string content in the collection, but there is no such data in flight movement, thus being ignored for further evaluation. Hashed index supports sharding and uses a hash function on a field value. The hashed field value is then used as a shared key in the cluster to partition data. In this experiment, there are no cluster or sharding, so the hash index is ignored for further evaluation.

Result with indexes

After finding the right combination of indexes to use on the system, the performance was measured once again in MongoDB. The performance gap from the tests with no indexes, ref Section 4.3.2, was quite significant, showing that indexes are crucial in any database systems. The performance of MongoDB with indexes can be seen in Figure 4.12.

Figure 4.12: MongoDB Performance with indexes.

Chapter 5

Comparison of Results

In this chapter, the compared results for RDBMS and MongoDB without and with indexes will be presented, as well as a comparison between them to look at how each of them performs on different queries. Based on the results that are presented in this chapter, a conclusion will be drawn in the next chapter.

5.1 RDBMS Improvement

The performance for RDBMS without and with indexes is quite significant, which can easily be visualized from Figure 5.1. By having indexes in the database, queries are sped up between 1,22 times in case of Beontra, which is the worst improvement, to 1840 times in the case of RTA Delay which is the best improvement. Beontra is one of those queries which is general and returns much data, while RTA Delay is specific and returns only a few rows at a time which impact the execution time. All other queries have been sped up significantly as well.

5.2 MongoDB Improvement

Almost the same story as with RDBMS can be told for MongoDB. The performance increased considerably after adding indexes on the database. The result from the comparison with and without indexes can be seen in Figure 5.2.

The query for Beontra was sped up by 14,46 times, which was the worst improvement, while RTA Delay was sped up by 2293 times. Most of the other queries were averaging a sped up by 110 times, but a few of them had even higher sped up.

Figure 5.1: RDBMS performance with and without indexes.

Figure 5.2: MongoDB performance with and without indexes.

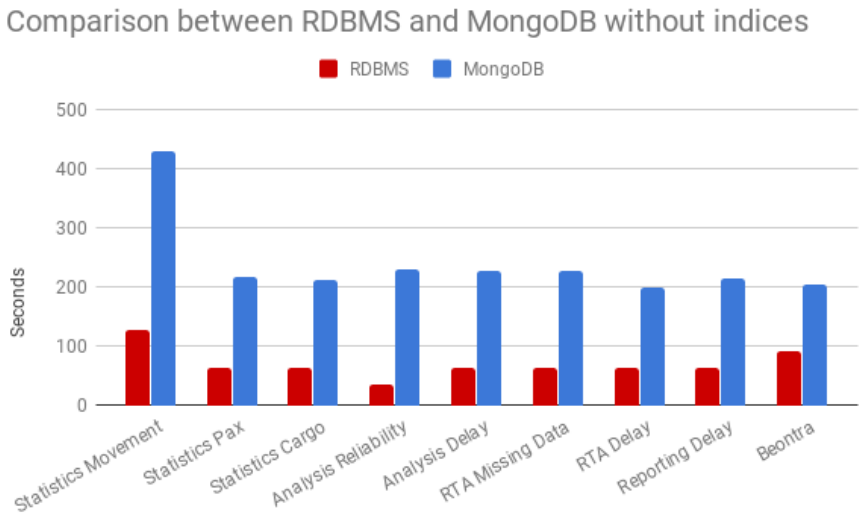


Figure 5.3: Comparison between RDBMS and MongoDB without indexes.

5.3 Comparison between RDBMS & MongoDB

Since the existing solution today in RDBMS is without any indexes, the same was done for MongoDB. The comparison between them can be seen in Figure 5.3.

There is a big difference between the two systems without any indexes. On average, the RDBMS uses 71seconds to execute and return results from a query. On the other hand, MongoDB uses an average of 241seconds to execute and return results from a query. This means that on average, the RDBMS is close to 3,4 times faster than MongoDB without any indexes.

After the benchmark of no indexes was completed, indexes were added on both database systems. The workload and queries remained the same. The result after adding indexes can be seen in Figure 5.4.

The performance gap between the two database systems is not as overwhelming as it was with no indexes but still contains significant variations. Some of the queries in RDBMS requires to join in order to fetch the needed results. These are Analysis Reliability & and Delay, Reporting Delay and Beontra. As can be seen in Figure 5.4, these queries have a higher execution time in RDBMS than in MongoDB. One of them is 1,75 times faster in MongoDB than in RDBMS, while another one is 14,15 times faster in MongoDB than in RDBMS.

The other queries do not require joins and perform more similar to each other than in the previous example. One of the queries performs precisely the same as each other, while another one is 2,86times faster in RDBMS than in MongoDB.

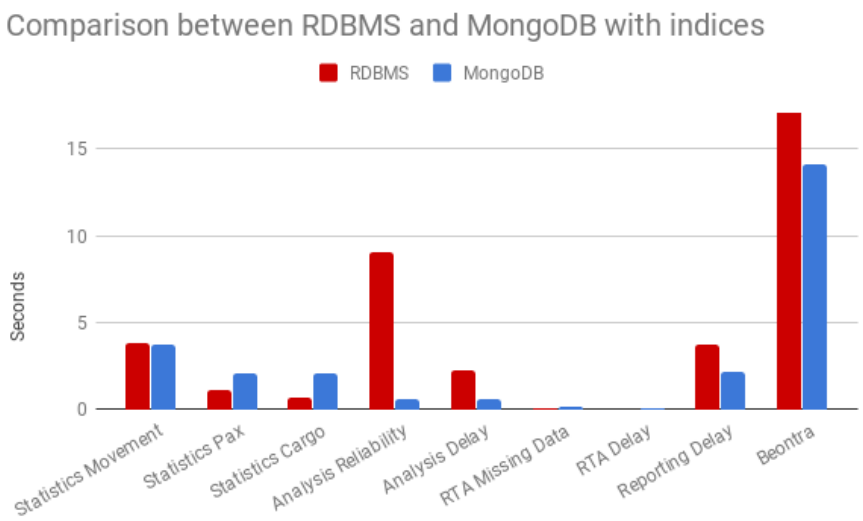


Figure 5.4: Comparison between RDBMS and MongoDB with indexes. The height of the y-axis has been cut in order to make variations clearer. Beontra in RDBMS is on 74 seconds which is way above the height of the graph.

5.4 Discussion of Result

Many cases might have influenced the result that was presented in the previous section. These will now be discussed and explained how their limitation is significant in the total picture of a live version of TRDB.

Caching of data:

During the experimental stages of this thesis, there have been a few fluctuations in terms of the execution time of queries. This is most likely due to caching of data and results from the two different database systems. Underway in the experiment, it was decided not to look further into this. One solution would be to clear the cache between each query, having the database systems on the same baseline. This, however, does not reflect any systems that would be in production for a large company. The real-life scenario is to have the database cache whatever it seems necessary and relevant. The database systems might have different approaches as to what and how to cache data. This makes it possible that one of them performs much better than the other one due to caching for specific queries, and should thus be taken into consideration when concluding.

Cold start vs warm start:

When measuring database performance, there is a difference between cold start and warm start. Cold start is done by executing the test before any of the data or indexes are cached

into memory. Warm start, on the other hand, is to run a few queries or a predefined set of queries before any tests, to make sure that the database has cached necessary index tables and data, as well as getting the CPU warm and ready for instructions.

A few queries were made before for both systems, and executed before any of the other queries was executed. The warm-up queries was a set of random queries, querying different kinds of data within a given time frame. The time frame that was used for the warm-up queries was the same as the use of case queries. The same time frame was chosen to represent a day to day usage of the databases, caching relevant indexes and data, and not old and stale ones. This decision, as well as the previous one about caching, was done to represent how a live system and database would work and function. A deviation in results would be likely if executed on a cold start.

Limited use cases:

The use cases that were defined in this thesis might be limited compared to what it could be in the future and the potential that could be extracted from TRDB. The use cases were described during meetings with Avinor and divided into different categories to have a spread of needed data to test the systems. Since the use cases are limited in terms of numbers, there might be queries that would run faster in MongoDB than in the RDBMS or the opposite. Even tho a conclusion has been made based on the result from the experiments, there might be exceptions to that. It would not be possible to cover everything. Based on the business understanding from Avinor and time constraints of this thesis, a sample of queries was made and results based on only that.

Data quality:

As already mentioned earlier, and also known by Avinor, the data quality in TRDB is not on an adequate level. There is missing data in the database, and it was found out that over 110 columns are never used and a few of the tables are empty. This causes particular problems for use cases and limits the potential of TRDB. Most columns and tables will eventually be filled with data, but as of now, this is not the case. The lack of data quality made it necessary to drop some of the wanted use cases outlined by Avinor. There is no doubt that the number of use cases will be increased when the data quality increases, making TRDB a more central part of the day to day operation at Avinor. The data quality has impacted the result and the number of use cases, making it a limitation in the experiment carried out in this thesis.

No live system:

The experimental setup that was created during this thesis is not a live system that receives updates like TRDB at Avinor. The actual system that Avinor uses frequently receives updates and depends on them. This was not possible in the experimental setup of TRDB due to many reasons like firewall, network, access problems, etc. It was therefore decided that the experimental setup of TRDB would be a local and closed database. This could have influenced the result in terms of order and prioritization of needed actions like reads,

writes and updates. However, this gave a more stable outline of the database since there were fewer variables to take into consideration.

No matching of data:

Since there are no received updates of flight movement in the experimental setup of TRDB, there is no matching of data as described in Section 2.1.3 of the real version of TRDB at Avinor. The matching process of updates can be time and resource consuming, especially when multiple fields need to be checked and verified. The RDBMS and MongoDB might have different methods of doing these updates and therefore, might impact the performance differently on a live system.

Key features not utilized:

To have the comparison as fair as possible, there where certain features of MongoDB that were not utilized, for example sharding, which was never done in this thesis. If the system would have been in MongoDB and in production, the chances are that these features would have been implemented early on. Depending on the shard key, workloads would have been distributed to several computers, eliminating the I/O cap from RAM or HDD. This is one of the characteristics that make NoSQL unique and make them able to handle large data sets. As said, it was not enabled to have the comparison as fair as possible and to test them on equal grounds.

As can be seen from the previous points, many objects might have influenced the result, and its limitation should be considered further if more work would be done on this topic.

Chapter 6

Conclusion and Future Work

This chapter will contain a conclusion based on the result and discussion that was presented in the previous chapter. Also, it will include a section of future work for what should be focused forward on.

6.1 Conclusion

In this thesis, we started by looking at the use cases of the transaction reporting database at Avinor, referred to as TRDB. These use cases were in advance outlined by Avinor and were meant to replace older systems. Experimental setup of an RDBMS TRDB was done on a local computer and queries were made based on the use cases that were defined. An alternative version of TRDB was created in MongoDB with the same data, and queries were converted to fit the structure and syntax of MongoDB. The same experiment was performed on both of the database technology in order to see which of them had the best performance in terms of execution time.

One of the problems that Avinor faces with TRDB is the execution time for queries that are being executed. To find out how the problem could be solved, a research goal and four research question were formed in order to address the research problem. The research goal and the four research question that was formed at the beginning of this thesis were as followed;

Goal: Improve TRDB considering execution time and overall performance for user queries.

RQ1: How is TRDB used and what data is important for usage?

RQ2: How can TRDB be improved in RDBMS with indexes?

RQ3: Which NoSQL system is most suited to replace the RDBMS of TRDB?

RQ4: Can a NoSQL alternative TRDB be as efficient or better than a solution in RDBMS?

For research question number 1, it was found out that TRDB was not in active use at Avinor due to the quality of data in the database. This was either due to incorrectness of data in AODB, which is a part the source of data, or the communication between AODB and TRDB was not sufficient. In addition to the quality of data, some data are missing or do not exist. Over 110 of the columns in TRDB's main table is always empty, and a few of the child tables are also empty. Due to the lack of data that is present, a few problems occur with the use cases that were defined since they depend on the data. Therefore some of them are not able to be implemented at this time.

It was also found out that many of the use cases depend on similar kind of data and does not require multiple joins. Based on queries that were made, there was never more than one join at a time, making them somewhat straightforward. It also showed that some of the data in the queries are more important than other and would be used more frequently, for example, like; flight departure date, latest version, AOBT, ATOT, delays and gate.

For research question number 2, a significant improvement can be made to the existing solution in RDBMS by adding indexes to the database. All of the queries was sped up, some more than others. The worst improvement was sped up by 1,22 times, and the best performing was sped up by 1840 times. Most of the other queries were averaging a sped up by 38,5 times; still a significant sped up on average. Based on the experiment in this thesis, most of the performance problem of TRDB in RDBMS can be fixed with correct indexes. There was only a handful of them made during the thesis, and more of them might be added based on the usage of data in the future.

For research question number 3, document-based database was chosen as the NoSQL technology. There are many document-based databases, but MongoDB was picked due to its popularity, enterprise support, scaling, replication and alongside many other features. Other categories that were evaluated in this thesis, but not as viable; Key-value, Graph, and Column Family databases.

For research question number 4, as can be seen in Figure 5.3, the performance between RDBMS and MongoDB without additional indexes are significant. The average execution time in RDBMS is 71 seconds, while the average execution time in MongoDB is 241 seconds. However, there are no operational databases that should not have indexes. Therefore, indexes were added to both databases, and the experiment was rerun.

In Figure 5.4, the result with additional indexes can be seen. From that figure, a final decision is hard to make since both databases perform better than the other one in some instances. It seems that queries that require join in RDBMS are generally faster in MongoDB than in RDBMS. The advantage with MongoDB is that there is no need for any joins since all tables have been denormalized. On the other hand, queries that does not require any joins seems to be faster in RDBMS than in MongoDB. Based on the result gathered in this thesis, it can be said that a NoSQL alternative, MongoDB, can more efficient than RDBMS in some instances, and slower in other cases. It is a viable option that would work and satisfy their use cases, but is it worth to swap the TRDB from an RDBMS to MongoDB as of today?

If a swap would be made, certain elements should be taken into consideration. The most significant and most crucial element is the cost. Such a swap would require many hours during the implementation of the database, integration towards other systems, creation of an application program interface which can be used to retrieve data, and thorough

testing of each step. The other element is the knowledge required to make such a solution viable. There is a big difference when working with an RDBMS and MongoDB, in terms of thinking and problem solving.

Based on these elements, along with the use cases that are defined in this thesis and the storage size that is needed, a swap from an RDBMS and MongoDB would not be recommended as of today. There are not sufficient advantages for such a swap, and it would cost more than what would be gained. If anything should be done, it is to focus more on the current solution in RDBMS. Improve the database with indexes and other techniques, which is mentioned in Section 6.2.1 and to focus on the data quality. There are many use cases, and more could be made, and TRDB could be a vital part of the operation for Avinor in the future.

That being said and concluded, will there ever be a time when MongoDB could be worth it? It depends on the future of TRDB. Since TRDB is holding historical data about all flights in Norway and Norwegian airspace, the scalability in terms of storage size might be a problem eventually. MongoDB is made to handle a massive amount of data and scales well, Section 4.1.5. In addition to scaling, it could also handle a higher workload due to multiple nodes, thus facilitating for new actors to use the data. This could, for example, be airlines, handlers at the airport, other airports, etc. This could be done through API's that limits the access to specific data, therefore, protecting sensitive information and business secrets that goes across different airlines in terms of passenger number, etc. If this would be the future of TRDB, large size storage and workload, MongoDB might be an alternative to further investigate in the future.

6.2 Future Work

In this section, a proposal of future work will be presented in terms of improvements that can be done in RDBMS to the already existing solution and what should be thought about when making changes. It also includes what should be done in order to get better results if such an experiment is to be repeated.

6.2.1 Future improvements of TRDB in RDBMS

There are further improvements that can be done to TRDB at Avinor to increase the performance of queries. As already concluded in this thesis, indexes speed up the performance significantly and should be added as soon as possible. If further improvements are required or wanted, materialized views and query tuning are examples of what can be done. Materialized view [46] are views that are updated periodically based on a query. If aggregation based queries are executed every month, the materialized view could be updated every day instead with aggregated data. Whenever a monthly query is executed, the database optimizer rewrites the query, using the materialized view instead if appropriate. This means that less than 32 rows would need to be aggregated in order to get the wanted result. The materialized views could be updated at the end of the day to reduce workload, or a trigger could be made to activate the aggregation.

Query tuning is done by rewriting queries in order to avoid specific keywords which can impact the performance in a negative way, resolve nested queries, make sure that the

query plan uses correct indexes, and update statistics in the database. There are times when keywords are needed and necessary, but sometimes it can be rewritten. Keywords such as *Distinct* and *Having* can lead to overhead, thus increasing execution time. Nested query or subquery is another example where the query optimizer performs less well and should be avoided if possible or rewritten. Queries might not use indexes if written wrong or in a different way, thus making it essential to write queries that utilize the indexes that already exists in the database. The last point is to update outdated statistics that can lead the optimizer to choose execution plans that perform worse than it would have with updated statistics. All these points are covered in-depth by Shasha and Bonnet [46]. The important part is that there are more improvements that can be done to the existing systems if wanted, but it needs to be worked on.

6.2.2 Use cases & Storage size

If this experiment is to be reproduced in the future, more use and test cases should be added. The result that was gathered during this thesis is not sufficient when taking a final decision. None the less, it gives a good baseline of how the two database systems compare to each other in terms of how it is wanted to be used. A statement could be made based on that, which was done in Section 6.1. In the future, the storage size would be increased, and it would be possible to see how the two database systems would compare in terms of scaling. Based on the result in this thesis and the possible result in the future, this could be done to have a better understanding of the scalability between the two systems.

Bibliography

[1] "CAP theorem with databases that "choose" CA, CP and AP — Download Scientific Diagram." [Online]. Available: https://www.researchgate.net/figure/CAP-theorem-with-databases-that-choose-CA-CP-and-AP_fig1_282519669

[2] "Delays – three questions and many answers — Eurocontrol." [Online]. Available: https://www.eurocontrol.int/news/delays-three-questions-and-many-answers

[3] "Databases and Collections — MongoDB Manual." [Online]. Available: https://docs.mongodb.com/manual/core/databases-and-collections/

[4] MongoDB, "MongoDB Architecture," *MongoDB*, 2018. [Online]. Available: https://www.mongodb.com/mongodb-architecture

[5] "What Is a Key-Value Database?" [Online]. Available: https://aws.amazon.com/nosql/key-value/

[6] "Graph Databases and Native Graph Technology — Neo4j." [Online]. Available: https://neo4j.com/blog/note-native-graph-databases/

[7] "SAP HANA Tutorial, Material and Certification Guide." [Online]. Available: http://www.saphanacentral.com/p/row-store-vs-column-store.html

[8] "Aggregation Pipeline — MongoDB Manual." [Online]. Available: https://docs.mongodb.com/manual/core/aggregation-pipeline/

[9] "Avinor Facts - Avinor." [Online]. Available: https://avinor.no/aviation/facts-and-data/avinor-facts/

[10] "Trafikkstatistikk - Avinor." [Online]. Available: https://avinor.no/konsern/om-oss/trafikkstatistikk/arkiv

[11] "Hvor mye flyr vi? — Avinor." [Online]. Available: https://www.ntbinfo.no/pressemelding/hvor-mye-flyr-vi?publisherId=17421123&releaseId=17580399

[12] "On-time performance for airlines and airports and Top 20 busiest routes," Tech. Rep., 2018. [Online]. Available: https://www.oag.com/hubfs/Free_Reports/Punctuality_League/2018/PunctualityReport2018.pdf?

[13] B. G. Tudorica and C. Bucur, "A comparison between several NoSQL databases with comments and notes," *Proceedings - RoEduNet IEEE International Conference*, pp. 1–5, 2011.

[14] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," *Proceedings of the 1st ACM symposium on Cloud computing - SoCC '10*, p. 143, 2010. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1807128.1807152

[15] A. Floratou, N. Teletia, D. J. Dewitt, J. M. Patel, and D. Zhang, "Can the Elephants Handle the NoSQL Onslaught?" vol. 5, no. 12, 2012. [Online]. Available: http://arxiv.org/abs/1208.4166

[16] Z. Parker, S. Poe, and S. V. Vrbsky, "Comparing NoSQL MongoDB to an SQL DB," p. 1, 2013.

[17] A. Boicea, F. Radulescu, and L. I. Agapin, "MongoDB vs Oracle - Database comparison," *Proceedings - 3rd International Conference on Emerging Intelligent Data and Web Technologies, EIDWT 2012*, pp. 330–335, 2012.

[18] S. Schmid, E. Galicz, and W. Reinhardt, "WMS performance of selected SQL and NoSQL databases," *ICMT 2015 - International Conference on Military Technologies 2015*, pp. 1–6, 2015.

[19] J. S. Van Der Veen, B. Van Der Waaij, and R. J. Meijer, "Sensor data storage performance: SQL or NoSQL, physical or virtual," *Proceedings - 2012 IEEE 5th International Conference on Cloud Computing, CLOUD 2012*, pp. 431–438, 2012.

[20] C. Hadjigeorgiou, "RDBMS vs NoSQL : Performance and Scaling Comparison MSc in High Performance Computing The University of Edinburgh," *RDBMS vs NoSQL: Performance and Scaling Comparison Christoforos*, 2013.

[21] "Airport AODB goes NoSQL (Part 1)." [Online]. Available: https://javadude.wordpress.com/2016/11/28/airport-aodb-goes-nosql-part-1/

[22] "Airport AODB goes NoSQL (Part 2) — The JavaDude Weblog." [Online]. Available: https://javadude.wordpress.com/2018/11/28/airport-aodb-goes-nosql-part-2/

[23] "Google Scholar." [Online]. Available: https://scholar.google.no/

[24] "Informasjonsløft frå Avinor - Vikebladet." [Online]. Available: https://www.vikebladet.no/naeringsliv/article7504833.ece

[25] J. Pokorny, "NoSQL databases: A step to database scalability in web environment," *International Journal of Web Information Systems*, vol. 9, no. 1, pp. 69–82, 2013.

[26] "The Essentials of Database Scalability: Vertical; Horizontal." [Online]. Available: https://blog.turbonomic.com/blog/on-technology/the-essentials-of-database-scalability-vertical-horizontal

[27] V. N. Gudivada, D. Rao, and V. V. Raghavan, "NoSQL Systems for Big Data Management," *2014 IEEE World Congress on Services*, pp. 190–197, 2014.

[28] S. Weber, "Weber_NoSQL_Paper.pdf," pp. 1–8.

[29] "NoSQL Databases: An Overview — ThoughtWorks." [Online]. Available: https://www.thoughtworks.com/insights/blog/nosql-databases-overview

[30] "Why NoSQL Databases is the Wrong Tool for Modern Application - MemSQL Blog." [Online]. Available: https://www.memsql.com/blog/why-nosql-databases-wrong-tool-for-modern-application/

[31] E. A. Brewer, "Towards Robust Towards Robust Distributed Systems Distributed Systems Inktomi at a Glance Inktomi at a Glance Company Overview Company Overview," Tech. Rep., 2000. [Online]. Available: http://pld.cs.luc.edu/courses/353/spr11/notes/brewer_keynote.pdf

[32] ——, "CAP Twelve Years Later: How the "Rules" Have Changed," Tech. Rep., 2012. [Online]. Available: http://dbmsmusings.blogspot.

[33] R. Zafar, E. Yafi, M. F. Zuhairi, and H. Dao, "Big Data: The NoSQL and RDBMS review," *ICICTM 2016 - Proceedings of the 1st International Conference on Information and Communication Technology*, no. May, pp. 120–126, 2017.

[34] M. Komorowski, "A history of storage cost." [Online]. Available: http://www.mkomo.com/cost-per-gigabyte

[35] "A Brief History of Non-Relational Databases - DATAVERSITY." [Online]. Available: https://www.dataversity.net/a-brief-history-of-non-relational-databases/#

[36] G. Harrison, "10 things you should know about NoSQL databases," Tech. Rep., 2010. [Online]. Available: http://techrepublic.com.com/2001-6240-0.html

[37] "Enterprise-Grade Support — MongoDB." [Online]. Available: https://www.mongodb.com/products/enterprise-grade-support

[38] "India to overtake U.S. on number of developers by 2017 — Computerworld." [Online]. Available: https://www.computerworld.com/article/2483690/india-to-overtake-u-s--on-number-of-developers-by-2017.html

[39] "ACI World: The voice of the world's airports." [Online]. Available: https://aci.aero/

[40] "Forside - SSB." [Online]. Available: https://www.ssb.no/

[41] "Lufttransport - kvartalsvis - SSB." [Online]. Available: https://www.ssb.no/transport-og-reiseliv/statistikker/flytrafikk/kvartal

[42] “Central Office for Delay Analysis (CODA) — Eurocontrol.” [Online]. Available: https://www.eurocontrol.int/articles/central-office-delay-analysis-coda

[43] “Average Delay per Flight on Arrival(mins) for December 2018,” Tech. Rep., 2018. [Online]. Available: https://www.eurocontrol.int/sites/default/files/publication/files/flad-december-2018.pdf

[44] “Flight Progress Messages Document,” Tech. Rep., 2017. [Online]. Available: https://www.eurocontrol.int/sites/default/files/publication/files/flight-progress-msg.pdf

[45] “B Operational.” [Online]. Available: http://www.beontra.com/products/b-operational/

[46] D. Shasha and P. Bonnet, *Database Tuning: Principles, Experiments and Troubleshooting Techniques*. Morgan Kaufmann Publishers, 2003.

[47] T. Kyte and D. Kuhn, *Expert Oracle Database Architecture*, third edit ed., 2014.

[48] R. Elmasri and S. B. Navathe, *Database Systems: Models, Language, Design and Application programming*, sixth edit ed., 2011.

[49] “Indexes and Index-Organized Tables.” [Online]. Available: https://docs.oracle.com/cd/E11882_01/server.112/e40540/indexiot.htm#CNCPT607

[50] “Optimizing the PL/SQL Challenge V: The Danger of Too Many Indexes — Oracle All Things SQL Blog.” [Online]. Available: https://blogs.oracle.com/sql/optimizing-the-plsql-challenge-v:-the-danger-of-too-many-indexes

[51] “The most popular database for modern apps — MongoDB.” [Online]. Available: https://www.mongodb.com/

[52] P. P. Srivastava, S. Goyal, and A. Kumar, “Analysis of various NoSql database,” *Proceedings of the 2015 International Conference on Green Computing and Internet of Things, ICGCIoT 2015*, pp. 539–544, 2016.

[53] K. Kolonko, “Master of Science in Software Engineering Performance comparison of the most popular relational and non-relational database management systems,” no. February, 2018.

[54] “DB-Engines Ranking - popularity ranking of database management systems.” [Online]. Available: https://db-engines.com/en/ranking

[55] “Our Customers — MongoDB.” [Online]. Available: https://www.mongodb.com/who-uses-mongodb

[56] “MongoDB System Properties.” [Online]. Available: https://db-engines.com/en/system/MongoDB

[57] “NoSQL Database — RavenDB ACID NoSQL Document Database.” [Online]. Available: https://ravendb.net/

[58] “Multi-model highly available NoSQL database - ArangoDB.” [Online]. Available: https://www.arangodb.com/

[59] “Big Data Architectures - NoSQL Use Cases for Key Value Databases — EMC.” [Online]. Available: https://infocus.dellemc.com/april_reeve/big-data-architectures-nosql-use-cases-for-key-value-databases/

[60] “Amazon DynamoDB - Overview.” [Online]. Available: https://aws.amazon.com/dynamodb/

[61] “Redis.” [Online]. Available: https://redis.io/

[62] “RocksDB — A persistent key-value store — RocksDB.” [Online]. Available: https://rocksdb.org/

[63] “Neo4j Graph Platform – The Leader in Graph Databases.” [Online]. Available: https://neo4j.com/

[64] “Graph Database — Multi-Model Database — OrientDB.” [Online]. Available: https://orientdb.com/

[65] R. Hecht and S. Jablonski, “NoSQL evaluation: A use case oriented survey,” *Proceedings - 2011 International Conference on Cloud and Service Computing, CSC 2011*, pp. 336–341, 2011.

[66] “Bigtable - Scalable NoSQL Database Service — Cloud Bigtable — Google Cloud.” [Online]. Available: https://cloud.google.com/bigtable/

[67] “Apache HBase – Apache HBase™ Home.” [Online]. Available: https://hbase.apache.org/

[68] “Apache Cassandra.” [Online]. Available: http://cassandra.apache.org/

[69] “What’s Unique About a Columnar Database? — FlyData.” [Online]. Available: https://www.flydata.com/blog/whats-unique-about-a-columnar-database/

[70] “Welcome to Python.org.” [Online]. Available: https://www.python.org/

[71] “Python Data Analysis Library — pandas: Python Data Analysis Library.” [Online]. Available: https://pandas.pydata.org/

[72] B. Practices, “A MongoDB White Paper RDBMS to MongoDB Migration Guide (White paper),” no. June, 2016. [Online]. Available: https://webassets.mongodb.com/_com_assets/collateral/RDBMStoMongoDBMigration.pdf?_ga=1.174593191.489481368.1471660179

[73] “The MongoDB 4.0 Manual — MongoDB Manual.” [Online]. Available: https://docs.mongodb.com/manual/

[74] “PyMongo 3.8.0 Documentation — PyMongo 3.8.0 documentation.” [Online]. Available: https://api.mongodb.com/python/current/

[75] “PYPL PopularitY of Programming Language index.” [Online]. Available: https://pypl.github.io/PYPL.html

[76] “cursor.explain() — MongoDB Manual.” [Online]. Available: https://docs.mongodb.com/manual/reference/method/cursor.explain/

[77] “Indexes — MongoDB Manual.” [Online]. Available: https://docs.mongodb.com/manual/indexes/

Appendix

I Traffic Report

Monthly Airport Traffic Statistics

CITY:	Alta			
AIRPORT/CITY:	Alta		IATA Airport Code:	ALF
Period (mmyy)	example: 0709 for Jul 2009		This Year	Last Year
I.	Aircraft Movements			
(a)	Passenger and Combi (combination) Aircraft	11,1%	570	513
(b)	All-Cargo Aircraft	−100,0%		6
(c)	Total Air Transport Movements (a + b)	9,8%	570	519
(d)	General Aviation and Other Aircraft Movements	38,6%	370	267
	Total Aircraft Movements (c + d)	19,6%	940	786
II.	Commercial Passengers			
(a)	International Passengers (enplaned + deplaned)		166	0
(b)	Domestic Passengers (enplaned + deplaned)	16,6%	31487	27001
(c)	Total Terminal Passengers (a + b)	17,2%	31653	27001
(d)	Direct Transit Passengers	21,6%	680	559
	Total Passengers (c + d)	17,3%	32333	27560
III.	Cargo (Freight & Mail) in Metric Tonnes			
(a)	International Freight (loaded + unloaded)	−100,0%	0,00	0,04
(b)	Domestic Freight (loaded + unloaded)	−16,7%	33,82	40,60
(c)	Total Freight (a + b)	−16,8%	33,82	40,64
(d)	Mail (loaded + unloaded)	−46,1%	7,76	14,40
	Total Cargo (c + d)	−24,5%	41,58	55,04
Event(s) affecting traffic for the period reported:				

Figure 1: Monthly Airport Traffic Statistics

II Data Structure

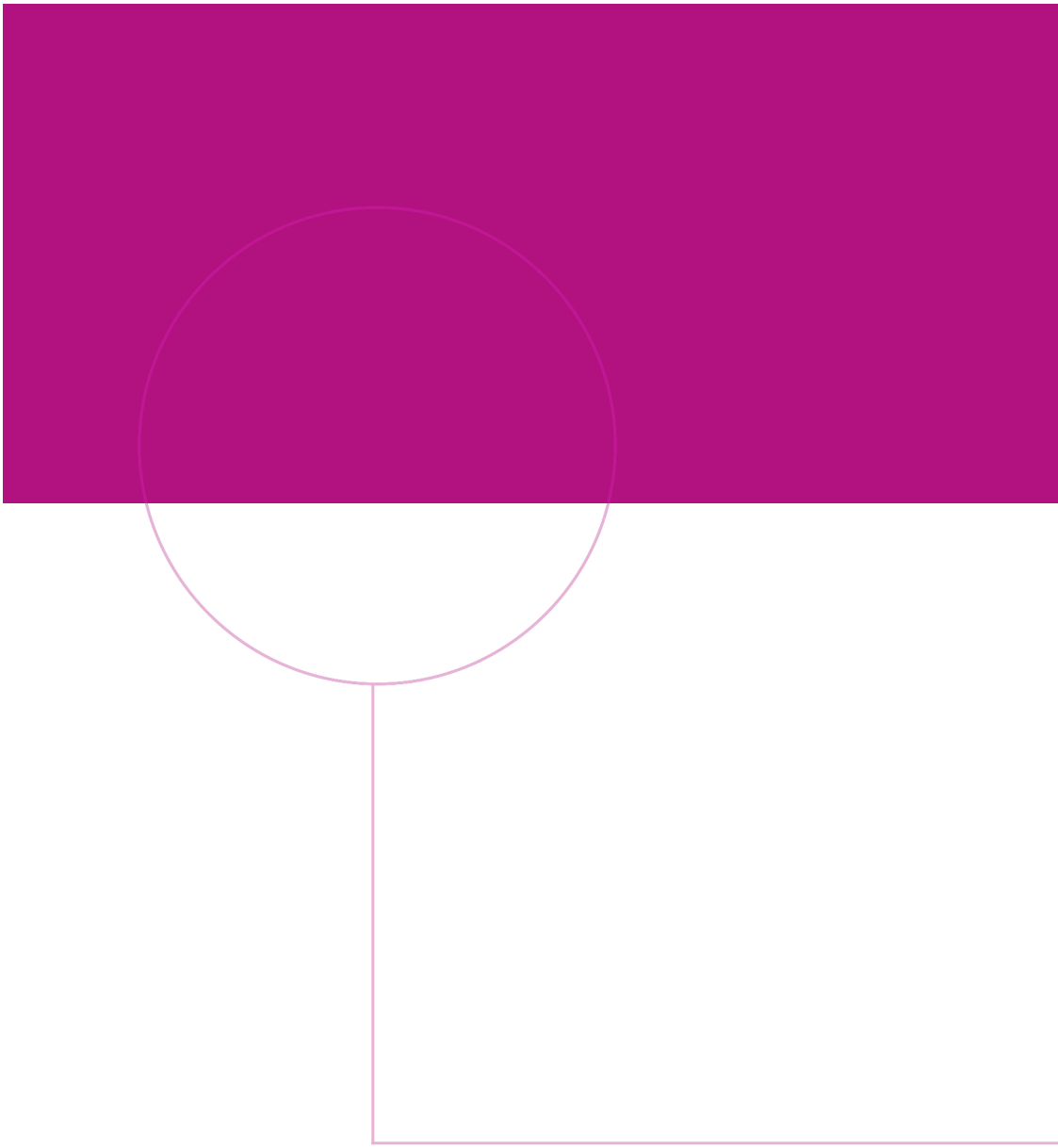
```
<FlightLegStateMessage xmlns="http://www.avinor.no/alti/...">
  <ambiHeader>
    <correlationId>CHR_01_2018-10-30_00:00:36.414</correlationId>
    <event>UPDATE</event>
    <messageType>FlightLegStateMessage</messageType>
    <messageVersion>1.0.0</messageVersion>
    <messageTimestamp>2018-10-30T00:00:38.587Z</messageTimestamp>
    <sourceOrganization>AODB</sourceOrganization>
    <sourceInterface>CHROMA</sourceInterface>
    <sourceTimestamp>2018-10-30T00:00:36Z</sourceTimestamp>
    <sourceResult>U</sourceResult>
  </ambiHeader>
  <flightLegDataChange>
    <flightLegIdentifier>
      <callsign>WIF866</callsign>
      <aircraftRegistration>LNWSC</aircraftRegistration>
      <flightId>WF866</flightId>
      <flightDepartureDate>2018-10-30</flightDepartureDate>
      <departureAirportIATA>BOO</departureAirportIATA>
      <arrivalAirportIATA>ANX</arrivalAirportIATA>
      <departureAirportICAO>ENBO</departureAirportICAO>
      <arrivalAirportICAO>ENAN</arrivalAirportICAO>
    </flightLegIdentifier>
    <aircraftData>
      <aircraftIATAType>DH2</aircraftIATAType>
      <aircraftICAOType>DH8B</aircraftICAOType>
    </aircraftData>
  </flightLegDataChange>
  <flightLegDataState>
    <flightLegIdentifier>
      <callsign>WIF866</callsign>
      <aircraftRegistration>LNWSC</aircraftRegistration>
      <flightId>WF866</flightId>
      <flightDepartureDate>2018-10-30</flightDepartureDate>
      <departureAirportIATA>BOO</departureAirportIATA>
      <arrivalAirportIATA>ANX</arrivalAirportIATA>
      <departureAirportICAO>ENBO</departureAirportICAO>
      <arrivalAirportICAO>ENAN</arrivalAirportICAO>
    </flightLegIdentifier>
    <aircraftData>
      <aircraftIATAType>DH2</aircraftIATAType>
      <aircraftICAOType>DH8B</aircraftICAOType>
      <aircraftNumberOfEngines>2</aircraftNumberOfEngines>
```

```
</aircraftData>
<passengerData>
  <paxSeatedOnBoard>0</paxSeatedOnBoard>
</passengerData>
<departureData>
  <departureSecurityIndicator>N</departureSecurityIndicator>
  <handlerDeparture>
    <handlerName>HANDLER_WGH_BOO</handlerName>
    <handlerServiceType>G</handlerServiceType>
  </handlerDeparture>
  <handlerDeparture>
    <handlerName>HANDLER_WGH_BOO</handlerName>
    <handlerServiceType>P</handlerServiceType>
  </handlerDeparture>
  <handlerDeparture>
    <handlerName>HANDLER_WGH_BOO</handlerName>
    <handlerServiceType>R</handlerServiceType>
  </handlerDeparture>
  <handlerDeparture>
    <handlerName>HANDLER_WGH_BOO</handlerName>
    <handlerServiceType>B</handlerServiceType>
  </handlerDeparture>
  <sobt>2018-10-30T13:55:00Z</sobt>
  <exot>P0Y0M0DT0H5M0S</exot>
  <linkedArrival>
    <aircraftRegistration>LNWII</aircraftRegistration>
    <flightId>WF853</flightId>
    <flightDepartureDate>2018-10-30</flightDepartureDate>
    <departureAirportIATA>SKN</departureAirportIATA>
    <arrivalAirportIATA>BOO</arrivalAirportIATA>
    <departureAirportICAO>ENSK</departureAirportICAO>
    <arrivalAirportICAO>ENBO</arrivalAirportICAO>
  </linkedArrival>
</departureData>
<arrivalData>
  <arrivalSecurityIndicator>N</arrivalSecurityIndicator>
  <sibt>2018-10-30T14:40:00Z</sibt>
  <linkedDeparture>
    <callsign>WIF866</callsign>
    <aircraftRegistration>LNWSC</aircraftRegistration>
    <flightId>WF866</flightId>
    <flightDepartureDate>2018-10-30</flightDepartureDate>
    <departureAirportIATA>ANX</departureAirportIATA>
    <arrivalAirportIATA>TOS</arrivalAirportIATA>
    <departureAirportICAO>ENAN</departureAirportICAO>
```

```
          <arrivalAirportICAO>ENTC</arrivalAirportICAO>
        </linkedDeparture>
      </arrivalData>
      <operatingAirlineIATA>WF</operatingAirlineIATA>
      <operatingAirlineICAO>WIF</operatingAirlineICAO>
      <operatingAirlineTicketed>WF</operatingAirlineTicketed>
      <flightDIIndicator>D</flightDIIndicator>
      <flightServiceTypeIATA>J</flightServiceTypeIATA>
      <flightServiceTypeICAO>S</flightServiceTypeICAO>
      <flightServiceTypeExtended>01</flightServiceTypeExtended>
      <numberOfAircraft>1</numberOfAircraft>
    </flightLegDataState>
</FlightLegStateMessage>
```

III Physical Storage of TRDB

Table name	Number of rows	Average size	Total size in B	Total size in GB
FLIGHT_LEG_DATA_TAB	17233858	489	8427356562	7.85
FLIGHT_LEG_ARR_DELAY_TAB	4123774	32	131960768	0.12
FLIGHT_LEG_BCROSSING_TAB	0	0	0	0
FLIGHT_LEG_CHECKIN_TAB	5392233	15	80883495	0.08
FLIGHT_LEG_CHR_BCROSSING_TAB	0	0	0	0
FLIGHT_LEG_CODE_SHARE_TAB	659812	24	15835488	0.01
FLIGHT_LEG_DEP_DELAY_TAB	3542975	32	113375200	0.11
FLIGHT_LEG_GATE_TAB	4988607	46	229475922	0.21
FLIGHT_LEG_HANDLER_ARR_TAB	29255641	29	848413589	0.79
FLIGHT_LEG_HANDLER_DEP_TAB	28428102	28	795986856	0.74
FLIGHT_LEG_LOAD_TAB	0	0	0	0
FLIGHT_LEG_REMARK_TAB	5470227	31	169577037	0.16
FLIGHT_LEG_TOUCH_AND_GO_TAB	0	0	0	0
Total			10812864917	10.7GB
FLIGHT_LEG_DATA_UPDT	15177039	357	5418202923	5.05
FLIGHT_LEG_ARR_DELAY_UPDT	467257	32	14952224	0.01
FLIGHT_LEG_BCROSSING_UPDT	0	0	0	0
FLIGHT_LEG_CHECKIN_UPDT	627565	17	10668605	0.01
FLIGHT_LEG_CHR_BCROSSING_UPDT	0	0	0	0
FLIGHT_LEG_CODE_SHARE_UPDT	857	24	20568	0
FLIGHT_LEG_DEP_DELAY_UPDT	453410	32	14509120	0.01
FLIGHT_LEG_GATE_UPDT	643882	36	23179752	0.02
FLIGHT_LEG_HANDLER_ARR_UPDT	270411	27	7301097	0.01
FLIGHT_LEG_HANDLER_DEP_UPDT	285946	27	7720542	0.01
FLIGHT_LEG_LOAD_UPDT	0	0	0	0
FLIGHT_LEG_REMARK_UPDT	267440	32	8558080	0.01
FLIGHT_LEG_TOUCH_AND_GO_UPDT	0	0	0	0
Total			5505112911	5.13GB



NTNU
Norwegian University of
Science and Technology

AVINOR