Optimizing data retrieval response time using localized database services

Optimera datahämtnings-svarstid med lokaliserade databastjänster

Philip Ekholm
Cem Lapovski

Degree: Bachelor 180 ECTS
Major: Computer Science
Program: Computer Engineering
Date for end seminar: 2019-05-24

Supervisor: Majid Ashouri Mousaabadi
Examiner: Radu-Casian Mihailescu
Abstract

Data processing is an aspect of system solutions which cannot be neglected. Response times is an equally important aspect in the modern online service. With cloud services reaching an all-time peak in popularity, it is important to know how to use them efficiently by providing the best results. The client, Axis Communications, is currently developing an IoT (Internet of Things) platform which is thought to deliver further software functionality and information in real-time to the end user. One of the elements of this service is event handling and processing. It requires events to be received and processed using AWS components before sending them through. In this thesis we aim to compare the performance between decorating an event stream using external services to decorating the same event stream with local services. In order to determine which of the solutions offer a better result for our client, we had to develop a new prototype of said solution. The process of development makes it possible to iterate and evaluate the prototype during the time of development. The evaluation should only be used as a guideline when considering event handling services as the number of parameters may differ and yield altered results. The results prove that our method for this solution had up to 5,16 times faster response time. The cloud-based solution also performed better when under heavy load by processing all of the requests simultaneously due to the scaling of AWS Lambda.

Sammanfattning

Acknowledgement

We would like to thank AXIS Communications for the opportunity to carry out this thesis. We would also like to thank Per-Anders Söderqvist at AXIS Communications for all the help and support with this thesis, this work would not be possible without him.
Glossary

**AWS** Amazon Web Services
**IoT** Internet of Things
**FIFO** First-In First-Out
**JSON** JavaScript Object Notation
**ARN** Amazon Resource Name
1 Introduction

More and more data is being generated in the world today. One of the major issues is processing large amounts of data in an efficient way. As a result, cloud services have begun to take a larger role in handling data [1]. However, when it comes to applying information to events being passed through a cloud service the situation for handling this can become tedious.

In this thesis we aim to better understand how much faster a 'local’ storage solution in AWS would differ from an opaque external database service.

1.1 Background

The amount of data generated in the world is increasing exponentially. A study shows that the amount has been doubling every two years and by 2020 we will have 10 times more data to process than we did in 2013 [2]. The data is increasing by such large amounts that traditional methods of data analysis are becoming inefficient. In order to analyze larger amounts of data, more automated and real-time approaches will be necessary to cope with this change.

More importantly, effective means of not only storing data but also access it is equally important and should not be neglected. There are many models for how to access data, many of these rely on linear searches of linear time complexity, something which is becoming more difficult to cope with. While the amount of data increases it can sometimes become difficult to access this data, especially depending on the architecture of access (e.g. hierarchical, relational). In the future it is most likely that large focus lies on being able to access data efficiently [3].

The fields of cloud computing are evolving from their dependencies on hardware operating systems. Businesses that are in the cloud computing genre have managed to capitalize on this burst and is also the reason behind its recent success. This slow but steady transition to cloud-based technology is also the reason for its fast evolution [4]. AWS is the leader in the commercial cloud computing market and is followed closely by Microsoft Azure and Google Compute Engine [5].

1.2 Problem

Axis Communications AB has sold many devices worldwide since they released their first network camera back in 1996. Just recently in 2017, the sales for video devices alone reached 8,602 billion SEK (about $925,8M) [6]. As the number of Axis devices that are connected to the internet keeps rising with new products and customers so does the need for maintaining all these products. Since the devices that are sold focus on security and surveillance, it is highly important that they have an active internet connection and are recording. Axis wants to give their customers the ability to upgrade and maintain their devices without expert help or knowledge using Axis Connect.

Axis Connect is an IoT platform that is under development by Axis Communications
AB. There are millions of Axis devices worldwide that are connected to the internet which are used for different purposes. When the service launches, the device owners will have a chance to opt-in for an improved experience. The service is a complement for those who seek to receive more functionality and information in real-time from their Axis devices. Some of the additional software functionality can be detailed crash reports, security hardening and health monitoring. The service will mostly be built on AWS for improved performance and keeping expenses low. Axis Connect provides a machine-to-machine API used by many different clients.

When clients accesses information regarding events, they want these to be as rich with information as possible (e.g. containing the name of folders, the ID and other associated information). We will be referring to this as decoration for the rest of this thesis. Axis Connect cannot perform the decoration service since in order to access such information a request would have to be made to an opaque external database service (see 2.3), whereas the response time for such events becomes too high at an increased throughput rate. Therefore, these events are currently undecorated. The main problem is how to make this process more efficient so the events could once more become decorated while still working efficiently, see listing 1 & 2.

Listing 1: Undecorated event

```json
{
    "folder": {
        "id": "123"
    }
}
```

Listing 2: Decorated event

```json
{
    "folder": {
        "id": "123",
        "name": "Camera Name"
    }
}
```

1.3 Research questions

Instead of having to rely on opaque database services which store a lot of varied information, it would be possible to create a 'localized' database containing only specific information that would be faster to access. In this context 'localized' refers to a cloud database which is tightly related to processing events. The research questions would then be:

- RQ 1: How would an event decorating system using a localized database service perform against an opaque database service, in terms of response time for decorating incoming messages?

- RQ 2: How would an event decorating system using a localized database service perform against a system merely sending through an event, rather than decorating it, in terms of response time for decorating incoming messages?

- RQ 3: Out of the possible options of creating this decoration (limited to 3, see 1.4), which would give the highest advantage vs. disadvantage ratio as a whole in terms of response time?
1.4 Limitations

The project is done on laptops that are running Ubuntu 16.04. These are provided by Axis. A requirement by Axis is that the project should be entirely done using AWS components, valid developer accounts will be given in order to achieve this. Because of this, other cloud services will not be taken into consideration for this thesis. Programming language are been limited to the following three: Python, Go and JavaScript.

Regarding the possible ways of solving the decoration problem, only three options are considered and evaluated. The three options that have been chosen have been narrowed down and specified from other possible options. These are evaluated in \[5.1\]
2 Theoretical Background

This section covers theoretical parts needed to understand later sections involving results. The concepts explained in this section include: A brief overlook of the Amazon Web Services used in this thesis, as well as opaque external databases.

2.1 Event sourcing

Event sourcing is a new take on persisting information. The traditional method requires that information is saved in a database model, usually done with SQL or NoSQL. With the traditional method, there is no way to tell when certain information was changed or what the previous state was. This is because, in order to update existing information, the current is usually erased [7]. Event sourcing handles this procedure differently by saving all the information as separate events. The events can be extracted to see what a state was at any point in time. The events are usually saved in a storage service in chronological order. Events are only added, never edited.

2.2 Amazon Web Services

Amazon Web Services is a provider of on-demand, server-less cloud computing for individuals, companies and governments. Amazon Web Services is a bundle of different cloud computing services that can be configured not only for a specific job but also to be used in conjunction with other services provided under AWS. AWS handles all server maintenance, updates and other related overhead tasks usually associated with setting up and renting dedicated servers, while offering a pay-as-you-go pricing model.

Different services and instances of these services can be accessed through the ARN (Amazon Resource Name), which is a unique ID for every instance of a service found on AWS. While configuring interaction between different services, usually the ARN is provided for this.

2.2.1 Simple Queue Service

SQS (Simple Queue Service) is a message queuing system that allows several messages to be received and processed at once [8]. This is done without creating a bottleneck or impacting system performance. It is possible to store, send and receive messages between AWS services. The messages received are guaranteed to be delivered within the AWS system as the messages are stored to improve redundancy. The service currently offers nearly unlimited throughput as well as unlimited messages in queue.

2.2.2 Simple Notification Service

SNS (Simple Notification Service) is a notification service with a built-in full publication and subscription support to other AWS services [8]. The service allows for publishing and posting messages to other devices through mobile push, email and SMS. When coupled with an SQS and Lambda it can perform complex tasks. The SNS can be divided into different topics which the subscribers can tap into and receive messages from. It is possible
to publish messages to the SNS from a Lambda, allowing it to be integrated to the flow of the system.

2.2.3 Simple Storage Service

S3 (Simple Storage Service) is an object storage service which offers high availability, security and performance [8]. It is also included in the broad AWS selection which means it too can be combined with the various other services. S3 is currently being used to store data from millions of applications from all around the world.

2.2.4 AWS Lambda

AWS Lambda is a cloud computing platform which allows for creating programs and executing these whenever triggered by an external event [8]. When an event has been created (e.g. an image has been uploaded to an S3-server) the Lambda can be activated to execute some written code in a preferred language (e.g. converting and cropping the image uploaded and then replace it in S3). If a function requires more processing power to keep up with the amount of data coming in, the Lambda will scale up accordingly to match this flow [9]. AWS remains responsible for handling the scalability of the platform.

2.2.5 DynamoDB

Amazon DynamoDB is a key-value and document database that delivers single-digit millisecond performance at any scale [8]. Unlike AWS S3 which is capable of handling unstructured data like a typical hard-drive, DynamoDB is used for processing structured data (tables). DynamoDB in its turn uses hash tables for accessing data, which in turn is faster than fetching unstructured data.

2.3 Opaque external database services

An opaque external database service is defined as a service whereas the parts comprising it are not visible, to what the term 'opaque' relates to. Furthermore, the service is not internal but rather external, such as a different cluster of servers need to be accessed in order to get relevant information, sometimes over long distances.

This type of service is usually performed in cases where you quickly want to store away data where the local infrastructure for this is insufficient. The event publishing service has since two years back, used different services for decoration which can be classified as unreliable. Due to the seemingly unstable access to information response times can vary from 200-5000 milliseconds, thus making asynchronous calls to this service expensive. Even though this service is opaque it is still measurable in performance which makes it worth analyzing for our thesis. The system architecture for this type of service is often simple, consisting only of a few components.
3 Related Work

This section covers works that have been done by others relevant to our thesis.

3.1 Query response time comparison NoSQL MongoDB with SQLDB Oracle

Humasak T. A. Simanjuntak et al. [10] have made comparisons in the response time between MongoDB and SQLDB Oracle. The first one is of a NoSQL type while the latter is of SQL type. Their findings were that most operations (select, insert, update operations) were faster on MongoDB than on Oracle, but the opposite when it came to deletions and more complex query operations.

3.1.1 Comments

This research is relevant since a comparison between two databases is made, with a large focus on response times, which in essence, is what this thesis focuses on. This research has not however made a comparison using AWS but rather relied on other database services.

3.2 Cloud Event Programming Paradigms

McGrath et al. [11] conducted two experiments that involve significantly different aspects of cloud computing. The two experiments that are presented are done primarily using AWS Lambda therefore the focus lies mainly on cloud computing and event handling using this service together with other AWS components. The article explains that cloud computing is being recognized by big actors in the business and that it has big potential. The experiments, Node.js is chosen as the runtime to leverage the Lambdefy library available specifically for that runtime. Lambdefy is a library for Node.js that can be used to make web applications run on AWS with little effort. The authors have shown varying results in form of performance statistics and a visualization on how the service is able to scale during heavy load.

3.2.1 Comments

This provides an insight to what AWS Lambda is capable of and how it performs during heavy loads. During our testing phase are pressuring the system with high amounts of messages and this proves that the service can handle it well. Furthermore, event handling is an aspect that we focus on which is also specified in this paper.

3.3 Type of NOSQL Databases and its Comparison with Relational Databases

Ameya Nayak, Anil Poriya and Dikshay Poojary [12] have made an overview of different NoSQL-databases that are commonly used and lists different advantages and disadvantages. They have also listed different characteristics, data models as well as query languages for operating these. Their conclusion was that NoSQL had a couple of advantages over relational databases, including but not limited to the ability to scale easier and are...
generally more flexible. There were some disadvantages however, related to no standard query language and no standardized way of interfacing NoSQL-services.

3.4 NoSQL real-time database performance comparison

Diogo Augusto Pereira, Wagner Ourique de Morais and Edison Pignaton de Freitas have made a comparison between common NoSQL-databases trying to determine which one is the fastest in a mix of operations, single thread as well as multi-thread [13]. The NoSQL-databases included in this comparison was MongoDB, Couchbase and RethinkDB. Their conclusion was that Couchbase performed best in most operations, with the exception for retrieving multiple documents and inserting documents using multiple threads, whereas MongoDB and RethinkDB performed better (with MongoDB marginally being faster than RethinkDB).

3.4.1 Comments

This paper is relevant not because of the comparison of different database-implementations, but rather in what kind of performance that can be expected from a similar implementation in terms of GET (and possibly PUT) operations.

3.5 Adaptive Time, Monetary Cost Aware Query Optimization on Cloud Database Systems

Chenxiao Wang, Zach Arani et.al have written a paper where they have looked at algorithms and techniques used for relational databases and come to the conclusion that these techniques are not designed for cloud databases, which typically (like AWS) operate on a Pay-as-you-go model where connections are being charged for [14]. In this paper they’ve develop optimization techniques which can save money by changing the type of query being made.

3.5.1 Comments

This paper is relevant since we’re developing a system that uses AWS-technologies, which charge for each time they’re being used. In order to minimize the cost these optimization techniques will be necessary to keep the cost low.
4 Method

In this thesis we have used a mixture of literature study and Nunamaker as a framework for our research [15]. The literature study has been chosen for evaluating which architecture would be best suited for building an efficient system for decorating events, while the Nunamaker has been chosen due to its systematical and iterative approach for building a system from a concept to a prototype. The Nunamaker process can be described as a five-step process as seen in figure 1 below.

4.1 Nunamaker & Chen’s process of developing a system

![Figure 1: The Nunamaker five step process described in a diagram. Starting off with a conceptual framework for how to design, to later develop a system architecture. After the first step, we design a system from which we analyze the output that leads to a working prototype from which we evaluate. The evaluation step focused mainly on accuracy of the division of data points. If any of these steps would fail it is possible to fall back and re-evaluate a certain stage.](image-url)
4.1.1 Constructing a conceptual framework

In the first part of the process we identify problems and investigate possible solutions for these. One systematical approach for identifying the problems is by building a problem tree. By building a problem tree a D&C (Divide and Conquer) approach can be performed against the problem.

The results from this step in the process are presented in 5.2.

4.1.2 Developing a system architecture, analyzing and designing the system

In these steps of the process an architectural view of the system is created with the different components of the system. Afterwards a conceptual model is created for the application. Typically, UML-documents can be used for this combined with other ways of systematically designing architectures.

The results from this step in the process are presented in 5.3.

4.1.3 Building a prototype system

The most important step to be able to make any performance measurements is to build an actual prototype system. This is done by using the developed system architecture as well as the problem tree.

The results from this step in the process are presented in 5.4.

4.1.4 Observing and evaluating the system

In the last step of the Nunamaker method a verification of the system is done. This can be done using formalized tests which confirms that all the functionality works as intended. This has been done by constructing tests which test different parts of the architecture to make sure the system works as a whole.

The results from this step in the process are presented in 5.5.

4.2 Selecting system solutions

In order to be able to select the best system solution, we must understand the differences between them. To achieve this, we conducted interviews with experts at Axis and also performed a literature study. The interviews are chosen to get an accurate and modern information from experts who are familiar with the system on a deep level. The literature study was chosen because we want to get a wide scope of how this type of problem can be solved using different methods. The purpose of this is also to achieve a better understanding of the subject at hand and research methods which are suitable for our thesis. We choose to keep the decision-making process to only two bases due to the amount of time that is available and we want information that is already tested and current.
4.2.1 Conducting interviews with Axis representatives

A number of interviews were performed with Axis representatives in order to gain knowledge on which architecture has been used in similar situations with the greatest success. This would in turn sum up against a decision to which option should be picked to implement. The conclusions from the interviews can be shown together with different empirical methods in ??.

4.2.2 Literature study

The main focus of literature study is aimed at understanding and increased accuracy for event sourcing. Due to the limitations we divide our searches into two main domains:

- Related works, frameworks and experiments regarding event sourcing.
- Methods and techniques for keeping latency and processing times low.

Our literature search has been focused mainly at the literature databases, IEEEXplore and ACM DL due to their relevance in this subject. Our search results have also been filtered in most cases to only display studies conducted in 2006 and onwards. This is mostly due to the release of AWS being the same year. We have filtered for mostly new results due to the risk of running over deprecated methods of implementation.

4.2.3 Interviews with experts

To better understand the system’s possibilities, vulnerabilities and limitations, we conducted interviews with experts at Axis. This is because they have previously implemented the opaque database service which is connected to the rest of the Axis system. Furthermore, they have been implementing solutions using the AWS environment and also have more experience within this scope. These are the questions that were asked during the interviews:

- Which successful implementation have been used for fetching information from an AWS DynamoDB or similar structure?
- What are the vulnerabilities for this service?
- What are the weaknesses for this service?
- What are the strengths for this service?
- Are there any other close options to consider with similar results?
5 Results

In this section we are covering the findings of which system solution would give the most benefits. After a decision is made regarding implementation, the Nunamaker method was used as a framework for planning, developing and evaluating the architecture which we developed.

5.1 Different system solutions for decorating events

The system consists of different types of event services that we have narrowed down to two types, the event and device service. The device service can be found on the majority of Axis devices and carries out the role of sending out events to the event service. The event service handles all incoming events and forwards them to the proper service needed for further processing or delivery. The decoration service is the system we are developing and processes the events by adding additional information and outputs them. Figure 2 is a side-by-side view between the previous system and our newly developed system.

![Diagram](image)

**Figure 2:** The previous system is displayed in the upper half. It requires the decoration service to fetch information from a database found outside of the service’s boundary. This is prone to increase latency times as the information must travel further. Our system can be seen on the lower half. It functions by having a ‘local’ database which is found closer to the decoration service. Because of the short distance it can perform queries faster than the previous system.
As mentioned in 1.4, the possible cases of building an optimized system for decorating events has been reduced to three options. These options are narrowed down and finalized with the help of a literature study as well as interviewing experts within the field. The options we have are concluding are the following:

- Option number I: The device service makes a request to the decoration service to fetch the decorations and add it to the event being decorated, before outputting the decorated event.
- Option number II: The event service makes a request to the decoration service to fetch the decorations and add it to the event being decorated, before outputting the decorated event.
- Option number III: The client fetches the decorations from the decoration service before outputting the decorated event.

5.1.1 Pros and Cons

After conducting interviews and using empirical methods we conclude different pros and cons regarding the different strategies in implementing an efficient decorating system.

The main advantage of (I) is that we have well decorated events all the way through the system. Option (I) comes with a lot of disadvantages however, including the need to implement functionality in many different services and setups which would be practically inapplicable. This approach would also create a lot of dependencies to other services which would otherwise not be required. With regard to highly reliable services minimizing the number of dependencies is critical.

In option (III) additional functionality would also have to be implemented in many different clients. Furthermore, many network mess requests would have to be performed to the services providing the information for the same event. This would have a bad effect on performance and many dependencies.

For option (II) it is possible to perform an event service fetch and adding decorations to every event that comes in, before processing it further. It could be possible to centralize this functionality and keep requests to one time per event. The problem with this approach is that it requires different synchronous calls to fetch the additional information which we consider bad for performance. If we were to localize this opaque request service however, we could remove the need for synchronous calls and make these internal with hash table access, hopefully significantly reducing response times.

5.2 Constructing a conceptual framework

In this section we gather information about how we would like the system to look like and try to predict eventual problems.
5.2.1 Problem breakdown

![Problem breakdown diagram](image)

*Figure 3: Problem breakdown, consisting of four main steps.*

A problem tree is created in order to properly address the problems needed to be solved in order to build a prototype. Our problem tree (see figure 3) consists of four branches which are further broken down in the following sections, with the exception of inserting data into database.

5.2.2 Lambdas

![Lambdas diagram](image)

*Figure 4: Problem breakdown of the Lambdas to be written.*

In summary there two Lambdas are written: one to pre-process the information and decorate the events. A second Lambda is also needed in order to process and extract necessary information from the events and add it to the database. Both of the Lambdas are necessary to complete the prototype.

5.2.3 Setting up services

![Services diagram](image)

*Figure 5: Problem tree for setting up services*
The service branch has three subbranches: Setting up Queue-services for handling data between different services, setting up an SNS to handle branching data between different services depending on the circumstances and also setting up DynamoDB with a table. See figure 5.

5.2.4 Establish connections between services

![Figure 6: Problem tree for establishing connections between different services](image)

In order to establish connections, several factors have to be taken into consideration. Proper authentication needs to be in place in order to allow for communication between the AWS components, since AWS by default does not allow programs/services to do anything. The necessary topics and subscriptions for the SNS to work properly also need to be setup. Finally, a filter needs to take place in order to limit the number of events being passed into the database-processor. In order to measure the systems performance under high pressure, we toggle the connection between two of the components. We refer to this in the results as **queue toggle**. These components are the preProcessor Lambda and eventInput SQS. This is done in order to simulate a batch of messages coming in at the same time. The connection between the can be toggled under settings in the AWS Lambda console.

5.2.5 Request sequence

In order for this system to work, a request sequence is built. This can be seen in figure 7, which gives an overview of the requests to be made in between the different services.
Figure 7: The conceptual requesting between different AWS-services. The system starts with some input from a user, script or other future potential service. Afterwards the event is forwarded to an input queue, from which the event is passed onto the preprocessor. The preprocessor fetches the folder name from the database and then decorates the request with this information. This is later passed onto the SNS, which checks specifically for a folder creation event, which in that case is passed onto the database. Either way, the event is passed onto the output queue from where a client can collect the data and see the end result.

5.3 Developing a system architecture, analyzing and designing the system

The previous system architecture in place consisted of fewer components, however we cannot rely on the results it provides. The high and volatile response times generated by the previous system are outdated. As described in 1.4 this thesis is limited to AWS. After a discussion with our supervisor it was concluded that a prototype environment would be setup in order to measure the performance difference. We looked at components related to AWS and setup up a system that can simulate the event-handler currently used by Axis. The general system layout can be seen in figure 8. The system architecture allows it to be expanded and inserted between two I/O points.
Figure 8: System overview of the AWS services used in the project. An input event (in our case sent by a local script) is sent to the input queue and later become processed by the preprocessor, later on sent through the SNS to the output queue from where the message can be collected. If a specific event with new information for decoration is processed, it will be added to another queue from where it will be inserted into the database.

5.3.1 eventInput

The eventInput SQS is the entry-point for any event that is to be 'fed' into the system. By using a queue, many different types of input can be accepted. Ranging from future services to manual input to script input, while maintaining the input data in a FIFO-structure.

5.3.2 preProcessor

The preprocessor in the architecture is responsible for taking in events, decorating these by calling the database for information and then pass these events on to the SNS. Traditionally this preprocessor used the opaque service instead of the localized database which is the main difference in the prototype from the system being used.
5.3.3 SNS

The Simple Notification Service is responsible for the following points:

- Forwarding any events received from the preprocessor to the event output, from where it can be collected.

- Filtering and forwarding specific events to a certain SQS from where the event info is extracted and inserted into the database. This is the case currently for folder create and update events from where a folder name is added or updated accordingly.

The information about a new folder being created and the name of it are sent to the appropriate SQS which in turn will add it to the database. It does also have one more topic for time measurement even if this is not an actual part of the system, see figure 8.

The SNS consists of a topic which two queues have different subscriptions to. The first one accepts any events passed onto the SQS, while the other queue has a filter implemented directly into the subscription. The preprocessor Lambda tags every folder event with a specific message attribute which the filter detects to make sure only these events end up in the secondary queue. Technically, more attributes could be attached but since deleting old events is not critical for this prototype this has not been implemented.

5.3.4 eventOutput

This is the final queue from which the end result can be seen and extracted either by script or manual output.

5.3.5 processToDB

This Lambda establishes a connection to the database service, extracts the ID and the folder name of the folder creation event and inserts this information as a new entry into the database. If the ID already has another folder name inside the database, the name is overwritten with the new name.

5.3.6 DynamoDB

The DynamoDB database service allows for structured data, which in this case keeps track of folder names and the ID which a certain folder is associated to. This information is kept in hash tables which makes adding and fetched items very fast (O(1)) [16].

5.3.7 timeCalc

timeCalc Lambda is a third service subscribed to the SNS which receives events and measures the time it takes for these to arrive. The Lambda is not connected to the eventOutput queue in order to reduce overhead in measurements mainly because this thesis focuses on database response time, not response time overall.
5.3.8 historicalEventsProd

Some of the old events can be found in an unstructured manner in the Amazon S3 service. In order to transfer some testing events, these are downloaded from historicalEventsProd, processed manually to some degree and then directly inserted into the DynamoDB. This is only done once to migrate old events into the new structure.

5.4 Building a prototype system

In this section we cover the steps taken to build a working prototype. The sections include setting up SQS-services, SNS-services including a filter for it, building the Lambdas, setting up DynamoDB as well as setting up the measurements. Common in all of the setups is that this can be done using the AWS online management console with a valid account.

5.4.1 Setting up SQS

Searching the AWS management console for SQS will show the service and from here an overview is shown of all current existing queues setup within a department. When initially setting up a new queue, the user is presented with a few options. In our case we are utilizing a standard queue with all of the standard settings. This is because the settings that are available does not give our thesis an advantage or disadvantage. Once the SQS has been created, it is ready to connect to the other AWS components.

5.4.2 Setting up SNS with filter

While working with AWS one can work against one single SNS, which is why it has not been more specifically named. A new topic was created for the preProcessor to publish the newly decorated event. One can create a new topic directly via the console, specify a name and be done with it. Then the preProcessor must be setup with publishing to the specific ARN which is specified in the script. No encryption or limited access was enabled.

After the topic has been created two subscriptions can be created to subscribe to that specific topic. The first subscription will accept anything that is passed onto the topic and no further action is necessary. The second subscription on the other hand requires a filter to be setup. By accessing subscription filter policy, the filter can be created. See listing 3.

Listing 3: The structure for allowing certain folder created events to pass through the filter into the DynamoDB. The event type must be of type created and also have a resource type of type folder.

```json
{
    "eventType": [
        "created"
    ],
    "resourceType": [
        "folder"
    ]
}
```
5.4.3 Setting up the Lambdas

The Lambdas for AWS are configurable with different parameters, some of these parameters include:

- **Role**: defines which permissions are granted to the Lambda.
- **Runtime**: defines what programming language is to be used.
- **Working memory**: defines how much memory each instance of Lambda should be assigned. This can impact processing time.
- **Timeout**: defines a time limit for how long the function can run before terminating the current execution.
- **Triggers**: defines whenever the Lambda will execute the written function.

Regarding roles, we are giving all of the Lambdas the same permissions. They have full functionality concerning read/write to the connected components, those being: SQS, SNS and DynamoDB. Regarding runtime, we are using Python 3.6 which offers many different libraries that we find useful for our thesis, see appendix A-C. Since AWS uses a pay-as-you-go model, the user is charged for the execution time in milliseconds for each instance of the Lambda. Therefore, it is critical that we keep our code as short and effective as possible as this will also yield good response times for the future potential clients which are going to use this service. Because of this we have to find the best price-to-performance ratio when determining the working memory. When choosing the timeout limit value, it is good practice to load test the function [17] as this will reveal an average execution time. The execution time is written after every execution, we used the default value of 3 seconds as a timeout limit. This limit is likely never to be achieved but it could be cost saving if written code should run into an infinite loop.

Building the preprocessor Lambda

For Lambda we want to process the incoming events and eventually decorate those that are missing the name field with such information. Firstly, we check if the event in question has any use for this information, this is done by reading the resource type and ID which are always provided. If the name field is missing, a request is sent to the database with the provided ID in order to retrieve the correct name. If the name is found in the database it will be attached to the event, if not, an empty name field is created and attached. Finally, the event shall be prepared with the processed information and published to the SNS.

The preprocessor Lambda is set at a working memory of 128 MB, which according to metrics should perform well without creating a bottleneck in the system. By increasing the working memory, we did not notice any significant or noteworthy improvements in our tests and therefore have decided to use 128MB to keep costs at a minimum. The batch size which is used for maximum events per batch is set to its highest limit of 10 for maximum throughput. See algorithm [I] for what the Lambda looks like.
Algorithm 1 prepProcessor

1: function MAIN(input) 
2:     for each event in input do 
3:         entry ← input.entryNumber               ▷ Only used when measuring time 
4:         if entry = first then 
5:             time ← Create timestamp         
6:         message ← event.message         
7:         type ← message.type         
8:         id ← message.id
9:         if message.name does not exist then 
10:            name ← name from database with (id)  
11:            message.name ← name         
12:         Publish to SNS with (message, type, entry)
13:         if time exists then 
14:             Send timestamp(time)           ▷ Done lastly so time measurement is not altered

This is the main piece for our system since it performs the major parts of the functionality required. It starts off by reading the input from the eventInput SQS, which can send messages in batches with up to ten messages at once [18]. The messages are then separated and processed individually. If we want to make time measurements, we make a timestamp after the event has been separated. The contents are then read and if the message does not contain a name-field a get request is sent to the database with the current ID which is found in every event. The request will return a name from the database if there is one available. Although if there is an occupied name field, the message will be sent along to the next step. When the name field has been occupied, either by decorating it or not, the message has reached the final step in this function and is prepared then published to the SNS.

Building the time calculating Lambda

The time measuring Lambda is working similarly to the preprocessor with dividing all of the messages which are received in batches of 10. When a message is received it will check for the information regarding queue number. If the message is the final one in the queue it will stop the stopwatch and calculate the time difference which will then be displayed in a readable format.

The time calculating Lambda is set to a working memory of 128 MB which is the minimum amount we can assign to it. We used this amount because we do not require this task to be performed quickly. It can therefore be assigned the minimum amount of working memory without any compromise. The batch size which is set to its highest limit of 10 for maximum throughput. See algorithm 2 for pseudocode.
Algorithm 2 timeCalc

```plaintext
function MAIN(input)
for each entry in input do
    entry ← input.entry.Number
    if event = last then ▷ Only used when measuring time
        Stop stopwatch
    Calculate time differences
```

Building the database handling Lambda

This Lambda has the responsibility to add new entries and update existing ones in the database. This will be done similarly to the preprocessor Lambda by firstly separating all of the incoming events. After this, the information is extracted and checked with certain parameters. If the event passes the conditions, it will put into the corresponding DynamoDB table, i.e. FolderNames.

The database handling Lambda is set to a working memory of 128 MB which is the minimum amount we can assign to it. We used this amount because we do not require this task to be performed quickly. It can therefore be assigned the minimum amount of working memory without any compromise. The batch size is set to its highest limit of 10 for maximum throughput. See See algorithm 3 for pseudocode.

Algorithm 3 processToDB

```plaintext
function MAIN(input)
for each entry in input do
    message ← input.message
    resource ← message.resource
    action ← message.action
    if resource = folder and (action = created or action = updated) then
        id ← message.id
        name ← message.name
        if name ≠ null and id ≠ null then ▷ Only add if name and id exist
            add id and name to database
```

5.4.4 Setting up DynamoDB

The DynamoDB requires little configuration to work for our purpose and is relatively easy to access. Tables can be created with as little as a name and a primary key, which is the main way of accessing an item in the table. Upon creation the database is set to a read/write capacity limit of 10 units [19]. When looking at metrics we concluded that we will need to increase this limit to "On-demand" to support our request peaks of 10,000 messages at once. If this is not done, it will slow down the performance in the preprocessor once requests reach a higher amount. The database table holds the structure of one primary key that is used for searching in the hash table and the folder name attribute linked to it. The table is easily extended with further attributes and sorting keys.
Figure 9: For our proof of concept we have used this simple structure of only one attribute being the folder name. The table can hold multiple attributes for the same item if needed. The IDs and folder names used in the preview are only for demonstrational purposes.

5.4.5 Access policy

For any service to have access to any other service on AWS, permissions are required. AWS handles these kinds of permissions via policies. The policies are written in a JSON format and follow a specific structure (specification of version, ID, statement, and to what extent), which can be added in a special permissions section in a service. Careful configuration is necessary for these services to work with each other, since by default they have next to no access to perform anything.
5.5 Observing and evaluating the system/Metrics evaluation

**Figure 10:** Measurements taken from using the opaque service. Starting from March 7th until April 4th. The response time is measured in milliseconds over the course of event processing. Starting March 28th, this service was used to decorate events being processed, which resulted in the volatile response times seen from the 28th, becoming increasingly volatile as per the number of events being stored in the service.

**Figure 11:** Response times using the decoration service. Ten measurements were gathered and both the average and median were taken from both. No queue toggle indicates that input events were sent into the service continuously, rather than all events at once.
Figure 12: Same as figure [11] but with a logarithmic scale along the Y-axis (base 10).

Figure 13: Response times but without using the decoration service, instead just sending the event through the preprocessor to see the amount of overhead. Ten measurements were gathered and both the average and median were taken from both. No queue toggle indicates that input events were sent into the service continuously, rather than all events at once.
**Figure 14:** Same as figure 13 but with a logarithmic scale along the Y-axis (base 10).

**Figure 15:** A comparison between the response times of our decoration service, the undecorated approach as well as the opaque service.
**Figure 16:** Same as figure 15 but with a logarithmic scale along the Y-axis (base 10).

**Figure 17:** The average of measuring the response time for 100 events being passed through 10 times.
**Figure 18:** The average of measuring the response time for 10,000 events being passed through 10 times.

**Figure 19:** The spread of measurements taken. The biggest variance from the average in the measurements can be confined to 15% from the average.
6 Analysis and Discussion

AWS is in the phase of expansion, with this comes new functionality and increased availability. Cloud services as a whole are being implemented more by each day that passes. This should be taken into consideration when reading the results for this thesis. The results are therefore just an insight of the current cloud service situation as of May 2019. Because of this there is a chance that researchers in this field will provide different results in the near future.

6.1 Related Works

Our thesis turns out to be quite unique in which not many related work have done the same kind of comparison which we perform. Nevertheless the related works are important in our thesis as guidance in what pitfalls to avoid and what lessons to be learned.

The paper *Type of NOSQL Databases and its Comparison with Relational Databases* [12] is important for our thesis since it shows what differs from the traditional relational databases and NoSQL-databases which we use in this thesis.

The paper *Adaptive Time, Monetary Cost Aware Query Optimization on Cloud Database Systems* [14] is useful in our thesis to minimize the cost of using AWS-services which use the Pay-as-you-go-structure with different optimization techniques. One of these include to being careful with what kind of query is being done, which can have a large impact on the cost of the service. The paper also discloses optimization techniques for optimizing lambdas to run on as little time as possible.

The paper *NoSQL real-time database performance comparison* is useful since it provides performance comparison between the different services, something which acts like a guideline in our comparisons for what is reasonable. A couple of hick-ups were sorted out in our system which affected performance thanks to this paper, including but not limited to the automatic throttle of the database after a certain amount of requests.

6.2 Evaluating different system solutions for decorating events

In the results different pros and cons are examined for which system implementation offers the most benefits over drawbacks from a software-architectural standpoint. While option (I) and (III) offer some promise option (II) seems to be the most (and possibly the only) viable option in our case. With this in mind we decide to go for option (II) since this option would at least theoretically cause the least amount of redundancy. A more careful evaluation and calculation of advantage and disadvantage ratio will not be necessary since the optimal option in our case is after this overview trivial.

6.3 Method discussion

By using Nunamaker & Chen’s method for system development, we had a guideline on how we are going to make progress during this thesis. Firstly, we conducted a study on how the system that we were to build. During this step, we interviewed Axis representatives whom
could provide insight and previous experience in similar system structures. The interviews also help us narrow down our options which lead us to construct a less complicated system architecture. This turns out to be a valuable step in the process because of the seamless integration and documentation available with AWS components.

6.4 Analysis of results

The aim of this thesis is to evaluate how a modern cloud-based solution would perform against a similar solution that uses a remote opaque database service, these results can be seen in 5.5. Tests are performed by sending events in bulk using batch sizes ranging from 1 to 10,000 messages with an exponential growth by base 10. The results in 5.5 prove that the solution that was entirely cloud-based had a better response times and also provided a better throughput than the latter. The performance is measured based on response times in regard to throughput per time unit. When using a 'localized' database on AWS, our results prove that in order to obtain low event response times, the overall processing time must be reduced.

Our service only checks for a matching ID-name pair in the database, if it is not found at the time of processing, the name property is ignored. This type of event indicates that the folder name has not been imported earlier and therefore it is impossible for it to be available in the database. The message is forwarded without processing, this is a requirement from our client in order to reduce response times and confusion from the end user.

When messages are delivered all at once, the processing time increased compared to if the messages were delivered continuously over time of execution to send the messages. Looking at figure [17] the processing times when messages were delivered over time matches the execution time to send the messages with an extraneously small difference. This is most likely due to the events being processed faster than our event service could send messages to the input queue. When the events were loaded into the queue and released at once, the decoration service was struggling a bit to keep up. The service was reaching times up to almost double the amount compared to the previous test which sent messages continuously over time. By calculating the time differences for the decorated event and opaque service in figures [17] and [18] we get an improved response time of approximately 5.16 times. When comparing the decoration service against the service not decorating, the result was not unexpected - the decoration took longer time since a database-call had to be done. By viewing figure [17] the decoration service took roughly 26% longer than the undecorated service. If you look at 10,000 events (figure [18]) this number was even higher - 33% when comparing against an undecorated service.
7 Conclusion

The research questions initially set were solved: the most effective means of performing the decoration task would be to send a request from the event service to the decoration service in order to fetch decorations and add these to the event before sending it on to the client. The results in figure 15 show large decrease in response time. Furthermore, the new architecture for decorating events by calling the ‘localized’ database service has also shown to be much faster in response time to undecorated scenarios rather than to opaque ones being approximately 5 times faster in response time to undecorated scenarios. This result is derived from taking how many messages that can be decorated using a localized service during the same time as decorating using the opaque service. Furthermore, decorating an event took roughly 30% longer rather than just skipping the decoration.

7.1 Contribution

As stated in 1.2 much effort has been put into attempts to evaluate and storing big data efficiently, but relatively few works have covered the ground of localizing information when fetching data remotely becomes unfeasible. This thesis also complements the existing publications made regarding comparisons of response times among SQL and NoSQL-databases. The results of this thesis can be used by Axis Communications for developing a more efficient system decorating events.

7.2 Future work

This thesis covers response time comparisons and evaluations for decorating events. For future work a similar implementation in other database-services could be done in order to compare not only response times but also differentiation in throughput, as well as general latency. As listed in 3.3 a similar implementation can be built into other NoSQL services.
References


A preProcessor.py

```python
import datetime, boto3, json

tsns = boto3.client("sns")
dynamodb = boto3.resource("dynamodb")
folderTable = dynamodb.Table("FolderNames")
timeTable = dynamodb.Table("TimeMeasurement")

def main(event, context):
    for e in event["Records"]:  
        entry = json.loads(e["messageAttributes"]['Entry'] ['stringValue'])
        if(entry == "first"):
            timestamp = datetime.datetime.now
            msg = json.loads(e['body'])
            resType = msg["resourceType"]
            uuid = msg['data']['current']['folder']['id']
            if ("name" not in msg['data']['current']['folder']):
                queryResponse = folderTable.get_item(Key={"id": uuid})
                try:
                    name = queryResponse['Item']['name']
                    msg['data']['current']['folder']['name'] = name
                except KeyError:
                    print("ID was not found in DB")
            sns.publish( 
                TopicArn="arn:aws:sns:eu-west-1:****:SendToQueues",  
                Message=json.dumps(msg),
                MessageAttributes={"resourceType": {  
                    "DataType": "String",
                    "StringValue": resType
                },  
                "actionType": {  
                    "DataType": "String",
                    "StringValue": msg["action"]
                },  
                "entry": {  
                    "DataType": "String",
                    "StringValue": entry
                }
            )
    putStartTime(timestamp)

def putStartTime(timestamp):
    currentDate = str(timestamp)
    timeTable.put_item(Item={"type": "Starting Time", "Current time": currentDate})
```
import datetime, boto3, json

dynamodb = boto3.resource("dynamodb")
timeTable = dynamodb.Table("TimeMeasurement")

def main(event, context):
    for e in event["Records"]:  
        entry = json.loads(e["Sns"]["MessageAttributes"]["entry"]["Value"])
        if(entry == "last"):
            putEndTime()
            putTimeDiff()

    def putEndTime():
        timestamp = datetime.datetime.now()
        currentDate = str(timestamp)
        timeTable.put_item(Item={"type": "Ending Time", "Current time": currentDate})

    def putTimeDiff():
        queryResponseStart = timeTable.get_item(Key={"type": "Starting Time"})
        queryResponseEnd = timeTable.get_item(Key={"type": "Ending Time"})
        try:
            startTime = queryResponseStart["Item"]['Current time']
            endTime = queryResponseEnd['Item']['Current time']
            startConvert = datetime.datetime.strptime(startTime, "%Y-%m-%d %H:%M:%S.%f")
            endConvert = datetime.datetime.strptime(endTime, "%Y-%m-%d %H:%M:%S.%f")
            delta = endConvert - startConvert
            diff = str(delta.seconds + delta.microseconds/1E6)
            timeTable.put_item(Item={"type": "Time Difference", "Current time": diff})
        except KeyError:
            print("Start/end time was not found in DB")
import boto3, json

dynamodb = boto3.resource("dynamodb")
folderTable = dynamodb.Table("FolderNames")

def main(event, context):
    for e in event["Records"]:
        newMsg = json.loads(e["body"])
        resType = newMsg["resourceType"]
        actionType = newMsg["action"]
        if(resType == "folder" and
           (actionType == "created" or actionType == "updated")): "updated"):
            try:
                uuid = newMsg["data"]['current']["folder"]['id']
                name = newMsg["data"]['current']["folder"]['name']
            except KeyError:
                print("ID or name property is missing")
            if(name is not None and uuid is not None):
                folderTable.put_item(Item={"id": uuid, "name": name})