Tuomo Moilanen

SCALABLE CLOUD DATABASE SOLUTION FOR SENSOR NETWORKS

Master’s Thesis
Degree Programme in Computer Science and Engineering
May 2013

ABSTRACT

This Master’s thesis presents the theory, design, and implementation of a scalable cloud database solution for a sensor network utilizing a cloud-based database-as-a-service platform. The sensor network is a part of the project called Internet of Things, a project studying on networking physical objects and visualizing of sensor data in a 3D environment. The challenge comes from being able to handle the massive data set expected to be produced by the sensor network. This is an example of the concept of Big Data, a data set so large that it becomes difficult and time consuming to be processed with conventional relational database methods. The solution can be found in NoSQL architectures. However, the distributed NoSQL architectures trade off several advantages of the relational database model to achieve database scaling, query performance, and partition tolerance. In this work, the database infrastructure is acquired as-a-service from one of the major cloud database service providers. Several decision criteria have been used to choose the database infrastructure service, such as scalability, absence of administrative duties, query functions, and service pricing. The work presents the rationale behind the database service selection and the design and implementation of the database and client software. Amazon’s DynamoDB is chosen as a target system due to its high scalability and performance, which offset its restrictions to query versatility. The resulting database design offering scalability beyond the traditional database management systems is presented with additional discussion on how to further improve the system and to implement new features.

Keywords: database architecture, Big Data, NoSQL, DynamoDB

Avainsanat: tietokanta-arkkitehtuuri, Big Data, NoSQL, DynamoDB
TABLE OF CONTENTS

ABSTRACT

TIIVISTELMÄ

TABLE OF CONTENTS

LIST OF ABBREVIATIONS AND SYMBOLS

FOREWORD

1. INTRODUCTION ......................................................................................................................... 9

2. BIG DATA ............................................................................................................................ 11
   2.1. Data Aggregation ................................................................................................................ 11
   2.2. Ethics and Privacy .............................................................................................................. 12

3. DATABASE MANAGEMENT SYSTEMS ................................................................................. 14
   3.1. Relational Databases ........................................................................................................ 14
   3.2. Database System Hardware Architectures ......................................................................... 15
   3.3. Database Scaling ............................................................................................................... 16
   3.4. Database Sharding ........................................................................................................... 16
   3.5. Database Concurrency Models ......................................................................................... 18

4. NOSQL .................................................................................................................................. 21
   4.1. NoSQL Categories ............................................................................................................ 21
       4.1.1. Key-value Store ........................................................................................................ 22
       4.1.2. Document Store ...................................................................................................... 22
       4.1.3. Wide Column Store ................................................................................................ 22
       4.1.4. Graph Store ............................................................................................................. 23
   4.2. Eventual Consistency ....................................................................................................... 23
   4.3. MapReduce ...................................................................................................................... 23
   4.4. Relational Database Management Systems Compared to NoSQL .................................. 26

5. CLOUD-BASED NOSQL DBMS SERVICES ........................................................................ 29
   5.1. DaaS Technologies Considered ....................................................................................... 29
   5.2. Google Bigtable ................................................................................................................ 30
       5.2.1. Data Model ................................................................................................................ 30
       5.2.2. Infrastructure and Architecture ................................................................................. 31
       5.2.3. Google App Engine ................................................................................................... 32
   5.3. Amazon Dynamo .............................................................................................................. 32
       5.3.1. Data Model ................................................................................................................ 33
### LIST OF ABBREVIATIONS AND SYMBOLS

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1NF</td>
<td>First Normal Form; a relational database normalization form</td>
</tr>
<tr>
<td>2NF</td>
<td>Second Normal Form; a relational database normalization form</td>
</tr>
<tr>
<td>2PC</td>
<td>Two-Phased Commit; a database concurrency technique to provide ACID guarantee</td>
</tr>
<tr>
<td>3NF</td>
<td>Third Normal Form; a relational database normalization form</td>
</tr>
<tr>
<td>4NF</td>
<td>Fourth Normal Form; a relational database normalization form</td>
</tr>
<tr>
<td>5NF</td>
<td>Fifth Normal Form; a relational database normalization form</td>
</tr>
<tr>
<td>ACID</td>
<td>Atomicity, Consistency, Isolation, Durability; a database concurrency model</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>AWS</td>
<td>Amazon Web Services</td>
</tr>
<tr>
<td>BASE</td>
<td>Basically Available, Soft state, Eventually consistent; a database concurrency model</td>
</tr>
<tr>
<td>BCNF</td>
<td>Boyce/Codd Normal Form; a database normalization form</td>
</tr>
<tr>
<td>CAP</td>
<td>Consistency, Availability, Partition tolerance; also known as “Brewer’s theorem”. Presents the trade-offs for distributed database management systems</td>
</tr>
<tr>
<td>CSV</td>
<td>Comma Separated Values; a file format</td>
</tr>
<tr>
<td>DaaS</td>
<td>Database-as-a-Service</td>
</tr>
<tr>
<td>DBMS</td>
<td>Database Management System</td>
</tr>
<tr>
<td>EMR</td>
<td>Elastic MapReduce; a service provided by Amazon Web Services</td>
</tr>
<tr>
<td>GAE</td>
<td>Google App Engine; Google’s platform-as-a-service</td>
</tr>
<tr>
<td>GFS</td>
<td>Google File System</td>
</tr>
<tr>
<td>GQL</td>
<td>Google Visualization API Query Language; an SQL-like query language for querying the App Engine Datastore</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HDD</td>
<td>Hard Disk Drive; a storage device based on magnetic discs</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol; an application protocol for distributed hypermedia information systems</td>
</tr>
<tr>
<td>HTTPS</td>
<td>Hypertext Transfer Protocol Secure</td>
</tr>
<tr>
<td>I/O</td>
<td>Input/Output</td>
</tr>
<tr>
<td>IoT</td>
<td>Internet of Things; a concept of physical objects forming data networks</td>
</tr>
<tr>
<td>IAM</td>
<td>Identity and Access Management; a service provided by Amazon Web Services</td>
</tr>
<tr>
<td>JSON</td>
<td>JavaScript Object Notation; a text-based open standard for data interchange</td>
</tr>
<tr>
<td>LINQ</td>
<td>Language Integrated Query; a .NET component providing querying capabilities</td>
</tr>
<tr>
<td>LSI</td>
<td>Local Secondary Index; an indexing feature of DynamoDB</td>
</tr>
<tr>
<td>MVCC</td>
<td>Multi-Version Concurrency Control; a concurrency control method</td>
</tr>
<tr>
<td>NoSQL</td>
<td>Not only Structured Query Language; a distributed and non-relational database management system schema</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>OLAP</td>
<td>Online Analytical Processing; a model for multidimensional analytical querying</td>
</tr>
<tr>
<td>OLTP</td>
<td>Online Transaction Processing; the traditional database transaction processing model</td>
</tr>
<tr>
<td>PaaS</td>
<td>Platform-as-a-Service</td>
</tr>
<tr>
<td>PACELC</td>
<td>Partition, Availability, Consistency, Else, Latency, Consistency; an extension to CAP theorem</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>RDBMS</td>
<td>Relational Database Management System</td>
</tr>
<tr>
<td>RFID</td>
<td>Radio Frequency IDentification; a technology to record the presence of an object through radio signals</td>
</tr>
<tr>
<td>S3</td>
<td>Simple Storage Service; a storage service provided by Amazon Web Services</td>
</tr>
<tr>
<td>SDK</td>
<td>Software Development Kit</td>
</tr>
<tr>
<td>SLA</td>
<td>Service Level Agreement; a part of formally defined service contract</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language; the standard relational database query language</td>
</tr>
<tr>
<td>SSD</td>
<td>Solid State Drive; flash-based data storage media</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol; a simple network protocol</td>
</tr>
<tr>
<td>URL</td>
<td>Uniform Resource Locator; a character string referencing to a resource</td>
</tr>
<tr>
<td>UTF-8</td>
<td>Universal Character Set Transformation Format - 8 bit; a variable width character encode</td>
</tr>
<tr>
<td>WGS-84</td>
<td>World Geodetic System; revision dated 1984</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language; a flexible, text-based format designed for web documents</td>
</tr>
<tr>
<td>YAML</td>
<td>YAML Ain’t Markup Language; a human-readable data serialization format</td>
</tr>
</tbody>
</table>
FOREWORD

I wish to express my gratitude to Professor Jukka Riekki for supervision, guidance, and feedback on this thesis, as well as Dr. Susanna Pirttikangas for supervising the thesis. In addition, I want to thank Timo Saloranta for presenting the challenge, thus identifying the subject of the thesis, and Iris Moilanen for proofreading the thesis.

Oulu, May 16th, 2013

Tuomo Moilanen
1. INTRODUCTION

The term *Internet of Things* (IoT) first emerged in early 1990s in a context of retrieving information describing RFID-tagged (Radio Frequency IDentification) object from a database, or an Internet service based on the object RFID. Since then, the idea has further evolved into a vision of a network encompassing everyday objects and entities, equipped with sensors, actuators, or other peripherals to measure and extend the potential applications and value of existing physical concepts. The increasing amounts of new, communicating concepts create dynamic environments, which are in turn connected to a network of networks – an Internet of Things. [1]

The IoT changes the people-originated nature of the information in the World Wide Web by introducing machine-published data and *thing-to-thing* communications [1, 2]. The vast amounts of real-time information streaming from the real-world objects all around create new challenges to how to handle the ever increasing flows of data, the Big Data. The most commonly used database model, the relational data model, is not an optimal solution to store and query the vast data sets generated by the increasing number of sensors in the network. The solution to scaling up to handle Big Data can be found with distributed server cluster architectures and schemaless data structures, the NoSQL (Not only SQL) technologies.

The intensifying data flows and the potential size of the aggregated data sets has to be taken into consideration when choosing the database management system (DBMS) infrastructure. This work describes the design and implementation of a database using a cloud-based Database-as-a-Service as a part of an IoT project. The general abstraction of the project system flow architecture is illustrated in Figure 1 (Saloranta T, personal communication, December 2012).

The network consists of sensors, which produce pairs of a timestamp and one or more values at variable frequencies to be stored into the database.
Each sensor has metadata that uniquely identifies the sensor and also contains other descriptive information such as location information and a measurement type.

This Master’s thesis presents the challenges and the process of design and implementation of a database solution to store sensor data for a sensor application that is a part of the IoT project. The database solution must be able to scale and handle the increasing amounts of sensors inserted into the network, and the increasing streams of data they produce. These large sensor data tables must then be available for fast and efficient querying. The work introduces the architectural design factors providing database scalability and reviews the scaling NoSQL Database-as-a-Services provided by the three large cloud service platform operators: Google, Amazon, and Microsoft.

Chapter 2 describes the concept of Big Data and its special characteristics and issues. It also reviews some of the technological advances and reasons that have led to exponentially increasing amounts of data generated in science, business, and other areas.

Chapter 3 presents an overview of general database structures starting from the database system server hardware architecture to different kinds of techniques used to scale the relational model and the database architecture to answer the needs of growing volumes of data and traffic.

The different types of NoSQL architectures are described in Chapter 4, and the strengths and trade-offs in comparison to relational database management system (RDBMS) model are evaluated. Chapter 4 also gives a quick overview on some of the essential technologies, such as MapReduce, used with several distributed database solutions.

Chapter 5 presents the three major cloud-based, NoSQL database-as-a-service providers: Google, Amazon, and Microsoft. This chapter describes the technological aspects, such as the database service’s architectures, data models, and characteristics, of Google’s Bigtable architecture and its service platform, Google App Engine, followed by Amazon’s Dynamo architecture and Amazon Web Services platform, and Microsoft’s Windows Azure service platform.

Chapter 6 discusses the applicability of each database service reviewed to the sensor network data storage scenario at hand and presents an expenditure analysis of a use case, followed by the decision of choosing the service platform for the application.

Chapter 7 presents the decisions behind the database design, and also covers some of the challenges and trade-offs required to create an efficient, working database solution. The chapter also presents the structure and implementation of the database client used to access the database, followed by query logic description and system testing.

Chapter 8 discusses some of the limitations of the system as well as potential database improvements to increase performance and to perform additional tasks using complementing services provided by the cloud platform.

Chapter 9 summarizes the general outline and the results of the thesis.
2. BIG DATA

Sensor networks produce an ever increasing amount of data. The generated data must be stored in a database and it must be available for querying for relevant data. However, as the amount of data grows, the performance of a relational database decreases, ultimately leading to unacceptable query times. When the size of the data set grows so large that it becomes too cumbersome to handle and work on, it is often called “Big Data” [3].

The definition of Big Data is intentionally subjective. It depends on the size and capacity of the application and also on the surrounding technology and techniques available. Big Data is usually used when the data sets’ size and complexity grows beyond the reasonable range of the commonly used database management tools and software, leading to unacceptable processing and managing times [4]. This is also what makes Big Data such a subjective concept as its definition depends on the available tools, techniques, bandwidth and processing power [5]. A smaller company may struggle and consider a set of few dozen terabytes to be Big Data, while a large enterprise might be handling several petabytes with a relative ease.

Technological advances in information technology have led to cheaper storage media and computing, making it possible to store, aggregate and process amounts of data previously unavailable or restricted to supercomputers [3, 4]. Similarly the scale of the data sets considered to be Big Data a few decades ago can be now fit into a normal hard disk drive available for consumers and can be handled by a personal computer [3]. Even though the actual size of the data set is the most visible feature, the defining features also include data complexity as a major contributor [5]. Big Data is, by its nature, networked data and, as such, the modeling of the relations has as important role as being able to handle the sheer size of the data [5]. These relations may not be that simple as the amount of entities to model grow and the connections increase, be it in an application for economical purposes or social networking [4, 5].

2.1. Data Aggregation

The first problems with Big Data arose when enterprises came to the conclusion that there was a need to create data warehouses to store and effectively query their historical business data for market analysis and reporting [6]. For a long time, the largest data sets were transactional data found in data warehouses, but new emerging concepts greatly affected the database environment. The more notable ones were the Web 2.0 technologies, such as wikis, blogs and social networks, generating large amounts of heavily networked data [7]. The vast amount of users producing immense amount of data, created new, significant streams of data in a scale that had been previously unavailable. These new data streams are also often highly relationally structured data, such as social networks, creating new challenges in data modeling and analysis.
The increasing and transforming sources of data streams have also affected the way scientific discovery is performed in many areas. The new methods of capturing and generating data often easily surpass the ability and capacity to process the data. Data generated by simulations and sensor networks, and data captured by high performance instruments have made data mining and visualization a significant part of the scientific process, leaving researchers often finding themselves lacking the tools and resources to efficiently process the inflating data sets. The existing solutions are often designed for a very specific task with no generalized, off-the-shelf solutions available, while the old analytical algorithms with high computational complexity result in unacceptable processing times. This new phase of scientific research moving away from the computational simulation phase has been established as the Fourth Research Paradigm: Data-intensive Scientific Discovery. [8]

The vast amounts of user generated content have also created new data gathering techniques such as crowdsourcing, where the data is collected from a large group of users (the “crowd”) submitting the data [3]. Another cause to the explosive increase in data storage sizes is the change in the nature of data: the ever increasing amount of video data produced and consumed [3]. One second of high definition video requires more than 2000 times the space to store in comparison to one page of plain text [3].

The efficient systems and techniques developed to process and perform analysis on large datasets, together with new, heavy streams of data and the development in computation power have led to a change in the fundamental approach to mining and studying datasets. No longer having to heavily compromise between the size of the dataset and the depth of the analysis, the traditionally data intensive analytics, such as business intelligence analytics and scientific research, offer new possibilities and applications [4, 9].

2.2. Ethics and Privacy

While the vastly inflated data sets offer new, more efficient possibilities for analysis, many of them may raise questions about the standards and policies. Security, privacy, intellectual property, and liability have become increasingly relevant as more and more services and companies record increasing amount of data. Unlike with physical goods, it often becomes unclear who in actuality owns the piece of data, or the right to utilize the data: the corporations and service providers aggregating the data, or the person that is the source of the data? What constitutes as a “fair use” of the data or who is responsible for negative consequences of inaccurate data? [3]

For many operators, the value of data is often governed by the trade-off between utility and privacy: high utilization datasets contain sensitive data that can lead to privacy issues, but on the other hand, increasing privacy reduces the validity of data, eventually rendering the dataset useless [10]. In addition, sensitive data stores like ones containing healthcare data or financial data are under additional legislation policies [3]. Recording and analyzing large amounts of traffic and trace logs to create behavioral pattern analyses to be used in advertisement placement (behavioral targeting) to show users advertisements they are likely to be interested in based on their previous browsing behavior, might raise concerns about consumer rights and privacy [11]. Similar concerns may be caused by companies utilizing and monetizing
the vast amount of personal information recorded by the social networks and selling the information to third parties.

Released datasets can be anonymized by utilizing various obfuscation techniques, such as generalization or bucketization, to protect privacy [10], but the sheer volume of the dataset may cause unexpected privacy issues through de-anonymization. Behavioral patterns and other unintended statistical data may be uncovered with thorough analysis from a large dataset, and the discovered knowledge may lead to a serious breach of consumer privacy. One well-known instance of such an incident was when a dataset of 20 million web search queries collected by AOL was released on the Internet in 2006, and it turned out that individuals could be singled out and identified from the dataset based on analyzing and cross-referencing the search data [12].
3. DATABASE MANAGEMENT SYSTEMS

The relational data model was first introduced in 1970 [13] to cater the needs of business database applications and transactions, and has stayed as the dominant data structure model in the database industry ever since [6]. Traditional database architectures are designed to run on a single computer [14] and follow the relational data structures enforced by relational scheme definitions, the categorized data normalization forms [15-17].

3.1. Relational Databases

Normalization is regarded as the general principle that is adhered to in the process of a relational database design. Normalization is a process to reform the data structure into refined, stable structures to maximize data accessibility and minimize data update anomalies [15-17]. The normalization process follows a set of data dependency rules categorized by normal forms. First Normal Form (1NF), Second Normal Form (2NF) and Third Normal Form (3NF) rule sets are common in practice and are supported by commercial tools while the higher forms of normalization Boyce/Codd Normal Form (BCNF), Fourth Normal Form (4NF) and Fifth Normal Form (5NF) are more academically inclined and less commonly implemented [15, 17].

Higher norms of formal normalizations provide many benefits including greater data integrity and stability to the data structures, but it all comes at the cost of decreased database system performance and more complex logical data access paths [15-17]. A large portion of performance degradation comes from the growing number of relatively expensive JOIN operations needed to aggregate data from the increasing amount of data tables mandated by the higher normal forms [15, 16]. Due to trade-offs with structural durability and system performance, finding a cost effective normalized structure becomes a problem, and database relations are often structured to follow the Third Normal Form (3NF) [17]. In other more performance intensive or partially unstructured data cases the database design may opt to reduce the level of normalization to increase performance, or even break the principles of normalized database design altogether to gain increased performance and simplified view of data through denormalization at the cost of data duplication, more complex data integrity rules and update anomalies [15, 16].

The traditional relational model was designed to provide functionality and performance to the operational online transaction processing (OLTP) applications, but it performs poorly on more extensive and complex analyses often performed on data warehouses. An OLTP modification, online analytical processing (OLAP) was developed to facilitate the analytical capabilities often required by business intelligence data mining. OLAP uses multidimensional data structures (OLAP cubes) consisting of numeric measures organized by dimensions. The nature of the data structure allows OLAP to provide multidimensional analytic queries with operations
focusing on aggregation management and re-orientating the multidimensional view of the data. [18]

3.2. Database System Hardware Architectures

When the volume of the data set grows, database system hardware architectures play a big role in how the database is able to be scaled. Database system hardware architectures can be categorized into three distinct architectures, from the oldest to the newest: shared-memory, shared-disk and shared-nothing [19].

The shared-memory multiprocessing architecture runs the database management system on a single node (computer) consisting of one or more cores sharing a common main memory and disk system [20]. The shared memory bandwidth creates a bottleneck, which severely limits the amount of cores the system can utilize [20]. Due to the hardware bottleneck, scaling a shared-memory system is difficult as the pure performance cannot scale past the common memory bandwidth limitations. The common database management systems using the shared-memory architecture include the popular MySQL, PostgreSQL, Oracle Database, and Microsoft SQL Server [20].

The shared-disk architecture on the other hand gives each node its own cores and main memories, but shares the common disk system between the nodes [20]. The shared-disk setup however suffers from some serious transaction management and scalability issues due to a private memory buffer pool combined with a shared disk storage requiring careful coordination of multiple copies of the same lock table and synchronizing writes to shared logs [19, 20]. This limits the amount of nodes to a small number making it a less popular choice to use for most database services, Oracle RAC being one of the few popular systems using it [20]. Shared-Disk is better suited for example enterprise mainframe level solutions, where mainframes have large processing capabilities and clustering a few can yield great benefits [21].

The shared-nothing architecture utilizes networking to create a cluster of individual, interconnected nodes that all have their private processors, main memories and disk systems [6, 20]. Coordination and data communication between nodes are carried out by messaging through the network [6] and the requests from the client are automatically routed to the node containing the resource [21]. Shared-nothing architecture is utilized by systems like Teradata, Vertica, ParAccel, Greenplum, Netezza and also nearly all the NoSQL engines run on shared-nothing [6, 20]. Shared-nothing systems usually perform partitioning (sharding) of data automatically to achieve divide-and-conquer type of parallelism, which introduces one of the major challenges with shared-nothing systems and scaling: load balancing [6, 19, 20].

Load balancing is a technique to divide the work load between the machines in the cluster (or other resources) to achieve the optimal utilization, throughput or response time [21]. Shared-nothing systems scale well only if the system’s load is balanced evenly across the nodes [20]. If one or a few nodes get a significant amount of data and traffic to overload a node, they create “hot spots” and the shared-nothing system’s performance degrades due to uneven load distribution [19, 20]. Properly balanced shared-nothing system should be able to scale until limited by the available network bandwidth [20].
3.3. Database Scaling

The first and simplest approach to increase the database system’s ability to process larger volumes of data is vertical scaling. Vertical scaling means that the raw performance of the database server is increased and this is usually achieved by upgrading the server machine’s processor or memory capacity [14]. However, vertical scaling cannot be utilized indefinitely as the expenses of upgrading increase exponentially, before reaching a cap when closing in to the current technology ceiling.

Horizontal scaling is the focal area where the traditional relational databases start to fail and where the chosen database hardware architectures come into play. Horizontal scaling approach means replicating and partitioning the data over several servers working in parallel, commonly referred as the cluster [22]. The scaling is done by simply adding new machines to the cluster to work in parallel to others [14]. The parallel pipelines allow the database system to handle large number of read and write operations [22] and the cluster can be run on a comparatively lower specification machines to reach the performance of a single vertically scaled server as the volume of the data increases [14]. This leads to horizontal scaling being usually a cheaper solution. Horizontal scaling comes with trade-offs as handling of a distributed server cluster adds another layer of complexity to administration, tolerances and data handling code in comparison to single server solutions [14].

3.4. Database Sharding

Even though traditional relational database architectures (shared-memory architectures) are designed to run on a single machine, there is a technique for the databases to scale horizontally and be able to handle larger volumes of data. This technique to scale a database horizontally is called “sharding”.

Sharding means basically partitioning the database into smaller pieces, shards, working in parallel to handle the growing load, essentially transforming a shared-memory architecture system into a shared-nothing architecture system [23]. Sharding has grown more popular during the past several years due to great increase in transaction and data volume processing it offers [23]. Sharding can be categorized into three different partitioning schemes: vertical partitioning, horizontal partitioning and directory-based partitioning.

Vertical partitioning is a simple partitioning scheme, where the data tables are split based on columns, making the vertical partitioning essentially a process of database normalization [23, 24]. Vertical partitioning approach is relatively straightforward to implement and has a low impact to the client or application design as a whole, but may be hard to scale if the DBMS requires further sharding due to growth [24].

Horizontal partitioning, also known as key-based or hash-based partitioning, refers to partitioning scheme by row data [23, 24]. The shard is designated by the key, or a part of it, or a hash function calculated using the key [14, 24]. A simple example with an incremental ID number would be calculating modulo operation between the ID number and the number of shards and use the result to distribute the data evenly
across the servers. This kind of approach, however, effectively fixes the amount of shards in the database cluster making it very difficult to scale further and doing so would often require a complete resharding of the data and altering the hash function [14, 24].

Horizontal partitioning can also be divided using ranges of values to designate shards. In range-based partitioning, the data is divided by some range of values, be it by an ID number range, a time range or some other range [24]. Range-based partitioning often suffers from uneven load distribution and hot-spots due to data or traffic [24]. Also, not having a uniform distribution with the chosen ranges makes resharding the overloaded shards with range-based scheme difficult [24]. Figure 2 illustrates a simple sharding based on the first letter of the primary key value.

![Diagram of Database Range Sharding](image)

**Figure 2.** Database range sharding based on the first letter of the primary key.

Directory-based partitioning consists of some of the more complex partitioning schemes. The schemes utilize a lookup service to run a central directory housing the partition scheme entry mapping or hash value ranges (consistent hashing) [14, 24]. This lightweight solution abstracts the partitioning scheme from the database access process, as the application queries the lookup directory to find out where the partitioning scheme has stored the required resource [14, 24]. This kind of approach makes further scaling or resharding doable without having to take down the service for the duration of the process [14, 24].

The sharding techniques bring out notable scalability benefits, but also create some additional problems. Most of the problems come from the change of environment from a single server database to a distributed server cluster. Records that were next to each other in the same table may no longer reside in the same table or even in the same server. Sharding brings in one major change to the usual SQL query statements: crippled JOIN expression [24, 25]. When the data is distributed across multiple servers, cross-server compilation queries quickly become very expensive, both processing and constraint wise, to perform as it can require multiple server queries instead of one for single server systems. [24]. A common solution for working around JOIN expressions is the denormalization of the data to remove the need for cross-shard compilation data [15, 24]. Denormalization can be achieved by replicating the data and storing the copy where it is needed to place all relevant data
in the same location. It should also be noted that this is a step away from the principles of normalized database design, bringing in the potential problems of denormalized data such as data inconsistencies and data integrity [15, 24].

Enforcing the relational data integrity of foreign keys, also known as referential integrity, across shards also becomes very difficult and taxing to manage, and most relational database management systems don’t support foreign keys across different nodes [24]. As the sharding transforms the DBMS into a shared-nothing system, it also brings in the challenges, like load balancing and administration, of the shared-nothing architecture [14, 23, 24]. If the data volume outgrows the capabilities of the sharded system or the system load becomes heavily unbalanced due to improper distribution scheme or uneven growth overloading a node, scaling or resharding while the database service is running might be impossible [14, 23, 24]. Depending on the chosen partitioning scheme, resharding might require the whole database to be processed for the data structure to satisfy the new partitioning scheme, leading to considerable service downtimes [14, 24]. Other schemes, like directory-based, might be able to perform resharding or scaling “on the fly” at the expense of increased complexity and potential points of failure at lookup or other management services [14, 24].

3.5. Database Concurrency Models

Traditionally, the reliability of a relational database system could be measured by its adherence to the ACID (Atomicity, Consistency, Isolation, Durability) transaction concurrency model [26]. Atomicity and consistency enforce the data integrity by eliminating the possibility to create invalid data structures in the database [27]. For a transaction to be atomic every single operation in a transaction must succeed or the whole transaction is rolled back [26, 27]. Consistency demands that the database is in a consistent state both before and after a transaction, thus requiring the transaction to conform the legal protocols [26, 27]. To satisfy the isolation constraint, each transaction must be independent and cannot interfere with other transactions, and similarly the durability attribute guarantees that once the transaction is committed, it can no longer be canceled or reverted [26, 27].

Even though ACID concurrency model is still widely used, the constraints it requires become problematic when coupled with distributed database architectures [27, 28]. A technique named 2PC (two-phase commit) protocol was developed to provide ACID guarantee across multiple database instances [27, 28]. 2PC protocol handles the commit of a distributed transaction in two phases [27, 28]. In the first phase the transaction coordinator logic requests each involved database instance for a confirmation whether or not a transaction commit is possible [27, 28]. If every instance responds with a positive acknowledgement, the coordinator asks each instance to commit, finalizing the transaction [27, 28]. 2PC protocol, however, has its own trade-offs. Processing of commit protocols, recovery from failure states and resource downtime from atomicity and isolation constraints all affect the overall DBMS performance, but the biggest impact is that the technique causes a hit on database availability [25, 27, 28]. 2PC protocol manages to achieve consistency in a distributed environment, but at the same time makes the solution less optimal if aiming for high data availability. [25, 27, 28].
A network partition results when a node in a distributed cluster can no longer communicate with another node due to crashing, loss of connection or some other malfunction [29]. In a case of a single server system, a one-node partition equals to server crash [29]. Partition-tolerance requires that operations terminate properly even if individual parts are failing and no less than a complete system network failure may result in an incorrect response [27, 29].

Brewer’s CAP theorem (Consistency, Availability, Partition-tolerance) (Figure 3) postulates that it is impossible for a database management system to achieve all three of the CAP properties [29]. The CAP theorem comes into play when dealing with distributed database systems due to the nature of having to forfeit either data consistency or availability [27, 29]. The consistency of data property follows the features of ACID concurrency model enforcing the validity and atomicity of the structured data [29]. Availability can be defined by every request received by a non-failing node in the system terminating in an intended response [27, 29]. However, the availability property does not define a hard deadline for how long the algorithm may run before termination [29]. If a database partition occurs, a distributed DBMS can either allow other nodes to continue working and no longer be synchronized with the failed partition, thus giving up data consistency, or make the data unavailable, thus giving up availability while preserving data consistency. Figure 3 exhibits how the traditional relational database management systems and several NoSQL systems place according to CAP division. BigTable, DynamoDB, and Azure Table Storage are further addressed in Chapter 5.

![Figure 3. Brewer’s CAP theorem for database management systems.](image)

An extension to the CAP theory, the PACELC model (Partition, Availability, Consistency, Else, Latency, Consistency), proposes a different taxonomy for data replicating systems [30]. The PACELC model stipulates that if a partition exists ("P"), how does the database system trade-off between availability and consistency ("AC"), and if a partition does not exist ("E"), how does the system trade-off between latency and consistency ("LC") [30]. Figure 4 further illustrates the logic of PACELC model. For example in a case of partition occurring, Amazon’s Dynamo architecture chooses to maintain availability instead of data consistency and loosens
consistency to achieve lower latencies under normal operational conditions (PA/EL),
while Google’s BigTable architecture elects to preserve consistency while sacrificing
availability and latency (PC/EC) [30].

Figure 4. Illustration of PACELC model classification.

To summarize, the ACID model provides the strongest semantics at the highest
cost and complexity for the DBMS, but its rigidness becomes a problem if a higher
availability is desired, as the ACID model enforces service unavailability over
relaxing structural data constraints [27, 31]. According to CAP theorem, consistency
constrains must be loosened for a system to obtain both high partition-tolerance and
high availability [25, 27, 29]. One solution is a model of eventual consistency, the
BASE paradigm: Basically Available, Soft state, Eventually consistent [27, 31].
Unlike the ACID model, the BASE paradigm focuses primarily on the high
availability of data instead of strong consistency or durability [31]. The BASE model
can tolerate stale data for a short amount of time as long as all instances achieve
consistency eventually [31]. Also data durability is sacrificed to gain increased
performance through soft states [27, 31]. A soft state is a system state that will expire
if the user does not refresh the state, and the states can be regenerated through some
additional computing or file input/output (I/O) [27, 31]. These characteristics result
in quick responses with the expense of some response accuracy lost due to possible
stale data and incomplete soft states [31].

High availability of BASE is obtained through supporting partial failures without a
total system failure, allowing the system to handle partial failures in clusters with
less complexity and cost [27, 31]. In other words, the system is available even though
parts of it may be unavailable. This kind structural elasticity and tolerance enables
the system to scale to levels unobtainable by ACID-enforced structures [27].
4. **NOSQL**

The concept of NoSQL was first introduced in 1998, only to be quickly forgotten and it was not until 2009 that NoSQL re-emerged with the development of several NoSQL solutions [32-34]. This makes NoSQL database solutions relatively young in comparison to the four decades of history of relational database management systems (Codd [13] introduced the relational data model in 1970). Defining the term “NoSQL”, abbreviated from “Not only SQL”, unambiguously is a bit of a tricky task. NoSQL is an umbrella term that, in theory, consists of everything not strictly adhering to SQL standards. In general, NoSQL database management systems are horizontally scaling, distributed shared-nothing systems that have no relational structure model or transactions, and use few or no SQL commands to query, modify, and store data in the system [22, 35]. The broad system divergence also leads to specifications of NoSQL systems being very diverse in comparison to RDBMS, which also translates into database management generally being performed through application programming interface (API) calls or a contextual language instead of standardized query language like SQL.

Many NoSQL database management systems were developed by the major information processing enterprises or services such as Google (Bigtable), Amazon (Dynamo), Facebook (Cassandra), Yahoo (PNUTS), Twitter (FlockDB) and LinkedIn (Voldemort) to cover their needs of large scale data storing and processing [32]. Many of the NoSQL architectures were originally developed for internal use to support the products and services offered, and later on made into a service platform product [22].

The Brewer’s CAP theorem states that a database system can achieve only two of the three properties [29]. Traditional RDBMS databases are systems that have consistency and availability, but have no partition tolerance [35]. Since NoSQL systems are distributed systems, a high partition tolerance is required [20]. This leaves systems to compromise between consistency and availability, and also deal with the aspect forfeited [20]. Amazon’s Dynamo loosens consistency to achieve high availability and partition tolerance, while Google’s Bigtable strives for high consistency and partition tolerance [35, 36]. The design philosophies compelled by the trade-offs have led to new approaches to database architectures.

### 4.1. NoSQL Categories

Taxonomy for NoSQL models is no less ambiguous to be defined as there is no official set of specifications [32]. Several different categorizations exist, but most often NoSQL systems are broken down into four core categories: key-value store, document store, wide column store, and graph store [32, 37]. Other divisions may include categories like object stores, grid & cloud stores, XML databases, multivalue databases, or several subcategories for key-value stores such as eventually consistent, hierarchical, ordered, and RAM (Random Access Memory) cached key-value stores [32].
4.1.1. Key-value Store

Key-value stores feature a very simple data structure model of a key and the associated bucket of values [22]. In a key-value store, sometimes called a tuple store, a single key-value index provides a simple and fast querying of data and high concurrency, but at the same time limits the available management operations to consist only of insert, update and delete operations directed at the values indexed by a single hash key [22, 33, 37].

Since the data bucket is accessed through a single key, everything apart from the indexed keys are opaque to the queries until their retrieval is called by the query results [37]. This forces additional queries to be executed via manually constructed and maintained secondary index tables containing relevant values for querying [37]. The secondary indices denormalize the database and they also require application developers and users to have knowledge of the database index structure [37]. Key-value stores include such as Riak, Redis, Voldemort, Amazon’s DynamoDB, and Microsoft’s Azure Table Storage [6, 32, 38].

4.1.2. Document Store

Document store model is quite similar to the key-value model, but is able to support more complex data in comparison to key-value stores at the cost of high performance read and write concurrency [22, 35]. A document database often stores data in semantic, semi-structured JSON (JavaScript Object Notation), XML (Extensible Markup Language) or YAML (YAML Ain’t Markup Language) format and offers a secondary index for more diverse querying [22, 33, 37]. Document stores allow a schemaless structure that doesn’t constrain tuple attributes to a preset structure, effectively allowing tuples to have any number of attributes, enabling dynamic data to be stored [22, 33]. Some popular document stores include SimpleDB, MongoDB, CouchDB, and Terrastore [22, 32].

4.1.3. Wide Column Store

Wide column stores, also often called extensible record stores or column-oriented stores, were inspired by Google’s Bigtable architecture [22]. Wide column stores still take the traditional table-based approach, but are accompanied by shared-nothing massively parallel processing with data compression and partitioning schema, enabling high performance and scaling [22, 35]. The scalability is achieved by a partitioning method that splits the table by both rows and columns, horizontally and vertically, over multiple nodes [22]. Rows behave in a similar manner to items in a document store and can be of variable length in attributes and fields. Rows are typically sharded through a range of primary keys, and columns are distributed over the nodes in pre-defined column groups [22]. Other wide column stores include such as Cassandra, HBase, Hypertable, and PNUT [22, 35].
Graph stores are designed to query and store graph data, and they represent interactions of heavily linked structures like social network or web site link structures [34]. Even though storing a graph into a SQL database is easy, querying it and especially traversing the graph can be very slow and inefficient [34]. A graph database works with three core abstractions to model the graph: a node, a relationship between nodes, and a key-value pair that can be attached to either a node or a relationship [37]. Graph stores include database systems like Neo4j and Yahoo’s FlockDB [32].

4.2. Eventual Consistency

The “E” of BASE, Eventually consistent, is an important aspect of parallel database system performance. The concept of eventual consistency was pioneered by Amazon’s Dynamo architecture as a method to achieve higher availability and scalability [22]. Consistency can be divided into two types of consistencies: strong consistency and weak consistency [25].

The strong consistency enforces a single storage image. Each subsequent query is required to return the updated value after an update operation completion [39]. Instead, weak consistency will not guarantee that a subsequent access returns an updated value. The time gap before an up-to-date value can be guaranteed is called an inconsistency window [39]. A weak consistency allows more efficient management and also offers the possibility of avoid using resource locks, which alleviates the problem of possible system locking [25, 39].

The eventual consistency model is a subtype of the weak consistency category. It gives up the enforced strong consistency by not guaranteeing that the read data is up-to-date [22]. The eventually consistent system propagates data updates to all nodes over time, making the system consistent if no new updates are performed to the object [22, 39]. Many systems also utilize other mechanisms like multi-version concurrency control (MVCC) or locks to provide a degree of consistency [22].

4.3. MapReduce

MapReduce is a software framework designed for parallel processing of large repositories in distributed systems and is an important component working in conjunction with NoSQL database management systems [6, 25, 40]. MapReduce algorithm was first introduced by Google to support the Google File System (GFS) handling large data sets in 2003 with other developers following in suit using Google’s papers on GFS and MapReduce as rough technical specifications [6, 40]. Prior to the development of MapReduce paradigm, special-computations had to be written and distributed to many computers to process large amounts of data for varying derived data [40]. Several issues like distribution, fault tolerance and
difficulties in parallelism were present [40, 41]. Now, MapReduce is an abstraction layer that presents a simple expression for computation. It hides the technical details such as managing, processing, parallelism and scaling underneath, offering a wide variety of applications ranging from data mining to machine learning [40, 41].

The MapReduce algorithm consists of two distinct primitives: map and reduce functions [40]. Map and reduce functions are modeled by

\[
\text{map}(k_1, v_1) \rightarrow \text{list}(k_2, v_2) \quad (1) \\
\text{reduce}(k_2, \text{list}(v_2)) \rightarrow \text{list}(v_3) \quad (2)
\]

where equations (1), (2) are the map and the reduce functions respectively [25, 40]. MapReduce takes a set of key-value pairs as an input and produces a compressed set of key-value pairs [40]. The map function, scripted by a user, produces a set of intermediate key-value pairs [40] and passes the intermediate products to reduce function. The reduce function, also written by the user, iterates through the intermediate results and merges the intermediate value associated with the same intermediate key to produce a smaller dataset [40]. Map and reduce functions can be easily parallelized to perform on large data sets and are adaptable to large distributed systems [25, 40]. Figure 5 (based on [40]) demonstrates the general MapReduce software framework execution flow.

![MapReduce execution flow diagram](image)

Figure 5. MapReduce software framework execution flow.

MapReduce execution in a distributed network cluster works by partitioning the data into a set of splits that are distributed to parallel map invocations [40]. A user specified partition function defines the intermediate key space, which is in turn used to distribute reduce invocations working in parallel across multiple machines in the network [40]. The amount of resulting output files correspond to the amount of reduce tasks defined, and the resulting output files are often utilized as input to
another application or the next chaining MapReduce call [40]. The numbered process labels in Figure 5 correspond to the following phases [40]:

1. The user program utilizing the MapReduce library divides the input file into fragments according to the configuration written by the user. The fragments typically are in the size range of 16 - 64 Mi a piece. MapReduce then initializes multiple instances of itself across a cluster of computers.
2. One program instance acts as a master node that assigns map and reduce tasks to idle worker instances.
3. A map worker instance reads the contents of the assigned input split and parses the key-value pairs. The values are then processed by the user-defined map function and the resulting intermediate values are stored into a memory buffer.
4. The buffered intermediate values are periodically written on the local disk and partitioned according to the user-defined partition function. The partition addresses are then supplied to the master node to be tasked to the reduce workers.
5. Reduce workers utilize remote procedure calls to read the intermediate data in the map worker’s local disks. After reading all designated partition data, the intermediate results are sorted in-memory to group identical keys together. In case the data partition is too large for in-memory sorting, external sorting is utilized.
6. After sorting, the reduce worker iterates through the intermediate result data. Iteration is utilized due to the possibility of large data partitions that are too large to be processed in-memory. During iteration, each unique intermediate key is passed, along with the corresponding intermediate data, to the user-defined reduce function that processes the intermediate data and appends the results to the reduce partition’s final output file.
7. After all map and reduce processes have been completed, the master node returns the MapReduce call back to the user application.

Another important aspect of MapReduce is its ability to handle errors. Since the processing clusters may consist of hundreds or thousands of computers working in parallel, fault states and hardware failures must be tolerated and recovered from. The master process periodically pings workers and in case of timeout, the worker is marked as failed [40]. Map and reduce tasks in progress on a failed worker are returned to idle state for rescheduling. Also, the completed map tasks on a failed worker have to be executed again due to intermediate data residing on a now inaccessible local disk of a failed machine [40]. Completed reduce tasks of failed workers are not vulnerable to re-execution since resulting output data files reside on a global file system [40]. This kind of dynamic process re-allocation makes MapReduce resilient even to large-scale node hardware failures. Other safety measures include atomic commits on map and reduce tasks and backup executions in case some of the last tasks still in progress take too long to finish, hinting on impending hardware failure or other error affecting the performance [40].

While Google’s MapReduce implementation is proprietary, an open source implementation of MapReduce is available through Hadoop project and many actors have developed their own implementations such as Amazon’s Elastic MapReduce (EMR) and Microsoft’s AzureMapReduce [38, 40].
4.4. Relational Database Management Systems Compared to NoSQL

With the rise of the Web 2.0, the very nature of data changed from the traditional transactional data to massive sparse unstructured data with loosely defined schemas [42]. RDBMS’s were not suited to handle the new, varied form of data [42]. The rigid, uncompromising pre-established table and relation structured architecture worked best when the available resources such as processing power, memory, disk space and network capacity were very limited, but are difficult to scale beyond a limit [42].

The unyielding relations, columns and indices of RDBMS make database scaling very challenging to perform while preserving serializable transaction semantics [20, 42]. The shared-memory system architecture also sets severe scalability limitations, although some shared-nothing SQL database systems have been developed (Greenplum, Aster Data, Vertica, etc.) [20].

In some cases, transaction overheads may become an issue. RDBMS has four major transaction overhead sources: logging, locking, latching, and buffer management [20, 43]. Logging creates a large overhead by writing everything on the disk twice: once on the database and once on the log. In addition the log must be forced to the disk to enforce durability, making logging very expensive [43]. A transaction must lock the resource before accessing it by setting a lock to the lock table to prevent dirty reads and writes [20, 43]. Latches, the internal structures used to ensure physical data integrity, generate overhead especially with serializing access to shared data structures in a multi-threaded environment [20, 43]. A buffer pool manages which set of fixed-size storage disk pages and records is in memory ready for accessing, generating a considerable amount of overhead [20, 43]. In comparison to the four major overhead sources for RDBM systems, the NoSQL systems generally have only two of the sources left: buffer management overhead from disk-based system as well as the drawbacks from multi-threading management [43].

Table 1 lists some of the differences between the traditional relational database management systems and NoSQL database management systems while Figure 6 (based on [44]) illustrates how the NoSQL databases position in a data complexity and data size graph in relation to each other and a typical RDBMS.

Boicea et al. [45] performed a series of tests to compare the performance between the traditional Oracle Database (SQL) and MongoDB, a popular open-source NoSQL document oriented store. The measurements were performed on objects of identical structure, each containing three fields of data. Insert, update and delete operations were processed in data sets varying in size from 10 to 100000 records. The size of the sample data set and the completion times for each database management system are listed in Table 2 (aggregated from tables in [45]) and the graphical presentation of the measurement data in Table 2 is illustrated in Figure 7. The results clearly exhibit Oracle Database’s huge increase in processing times for the larger datasets while MongoDB handles the operations in a fraction of the time it takes Oracle Database to complete the processing of the dataset. Especially with the update and delete latencies, the increased dataset size was nearly unnoticeable in processing times for MongoDB, unlike the steeply escalating latencies for Oracle Database. [45]
Table 1. RDBMS and NoSQL database comparison

<table>
<thead>
<tr>
<th>System</th>
<th>RDBMS</th>
<th>NoSQL databases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>Centralized: shared-memory</td>
<td>Distributed: shared-nothing</td>
</tr>
<tr>
<td>Data Model</td>
<td>Relations: the database is viewed as a whole</td>
<td>All entities are independent data units and can be freely relocated</td>
</tr>
<tr>
<td>Schema</td>
<td>Structured</td>
<td>Unstructured or semi-structured</td>
</tr>
<tr>
<td>Transaction model</td>
<td>ACID</td>
<td>BASE</td>
</tr>
<tr>
<td>Serializable</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>CAP division</td>
<td>CA</td>
<td>AP or CP</td>
</tr>
<tr>
<td>Scaling</td>
<td>Difficult</td>
<td>Scales easily horizontally</td>
</tr>
<tr>
<td>Querying</td>
<td>SQL standard</td>
<td>Ranging from simple API lookup to more flexible document and graph model queries</td>
</tr>
<tr>
<td>Query flexibility</td>
<td>SQL</td>
<td>Varies from graph queries to single index fetches</td>
</tr>
<tr>
<td>Ad-hoc querying</td>
<td>Easy</td>
<td>Difficult</td>
</tr>
<tr>
<td>Capacity provisioning</td>
<td>Statically provisioned</td>
<td>Dynamically provisioned</td>
</tr>
<tr>
<td>Design focus</td>
<td>Online atomic transactions</td>
<td>Web 2.0 applications</td>
</tr>
<tr>
<td>Administration</td>
<td>Lower</td>
<td>High (distributed system)</td>
</tr>
</tbody>
</table>

Figure 6. Data complexity and data size graph for NoSQL and RDBMS.
Table 2. Performance comparison between Oracle Database and MongoDB

<table>
<thead>
<tr>
<th>No. of records</th>
<th>Insert times (ms)</th>
<th>Update times (ms)</th>
<th>Delete times (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Oracle</td>
<td>MongoDB</td>
<td>Oracle</td>
</tr>
<tr>
<td>10</td>
<td>31</td>
<td>800</td>
<td>453</td>
</tr>
<tr>
<td>100</td>
<td>47</td>
<td>4</td>
<td>47</td>
</tr>
<tr>
<td>1000</td>
<td>1563</td>
<td>40</td>
<td>47</td>
</tr>
<tr>
<td>10000</td>
<td>8750</td>
<td>681</td>
<td>94</td>
</tr>
<tr>
<td>100000</td>
<td>83287</td>
<td>4350</td>
<td>1343</td>
</tr>
<tr>
<td>1000000</td>
<td>882078</td>
<td>57871</td>
<td>27782</td>
</tr>
</tbody>
</table>

Figure 7. Performance comparison between Oracle Database and MongoDB.
5. CLOUD-BASED NOSQL DBMS SERVICES

One of the requirements for the database system designed in this thesis was that it will be deployed into a hosted, scaling cloud database service. Currently, around 150 different NoSQL database management systems exist with a great majority of them being open-source systems [46]. While the open-source software culture brings many implementations available for deployment, the prevalent non-proprietary model, together with the relative immaturity of the NoSQL database scheme, has also a side effect of only few systems being provided as a Database-as-a-Service (DaaS).

5.1. DaaS Technologies Considered

The DaaS technologies of three large cloud service providers, Google, Amazon, and Microsoft, are reviewed. All three providers maintain a cloud service platform with several different application services, including NoSQL database solutions. Google features the BigTable-based App Engine Datastore, a wide column store, while Amazon Web Services provide DynamoDB key-value store and an older, non-scaling SimpleDB document store. Microsoft’s NoSQL solution is the Azure Table Storage key-value store on Windows Azure Cloud (Table 3). Figure 8 illustrates how the reviewed services distribute according to CAP and PACELC models (Chapter 3.4.).

<table>
<thead>
<tr>
<th>Service provider</th>
<th>Cloud service platform</th>
<th>NoSQL database service</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Google</td>
<td>Google App Engine</td>
<td>App Engine Datastore</td>
<td>Wide column</td>
</tr>
<tr>
<td>Amazon</td>
<td>Amazon Web Services</td>
<td>DynamoDB</td>
<td>Key-value Document</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SimpleDB</td>
<td></td>
</tr>
<tr>
<td>Microsoft</td>
<td>Windows Azure</td>
<td>Azure Table Storage</td>
<td>Key-value</td>
</tr>
</tbody>
</table>

Figure 8. Reviewed DaaS distribution according to CAP and PACELC models.
5.2. Google Bigtable

Bigtable is Google’s proprietary distributed database management architecture designed for managing structured data while providing high scalability and performance. Many of Google’s own services, such as Google Earth, Google Analytics, Orkut, and web indexing, utilize Bigtable as the underlying data storage [47]. Bigtable is also designed with flexibility in mind to cope with the very diverse demands placed on the system data size, ranging from URL (Uniform resource locator) strings to satellite imagery [47].

5.2.1. Data Model

Bigtable is a “sparse, distributed, persistent multi-dimensional sorted map [47]” that is indexed by a row key, a column key, and a timestamp. The table is ordered by a row key and partitioned dynamically by a row range into tablets, which are the units of distribution in database load balancing. The row keys can be arbitrary strings and have a maximum size of 64 KiB, and the operations to row keys are atomic. The row key range partitioning can be utilized further in the database and client design to limit the communication caused by shorter range queries to a small amount of machines in the cluster. [47]

Column families are created by grouping columns into sets by their key prefixes and the column family keys have a syntax of family:qualifier. Column families have several properties relevant to the functioning and performance of the database, and they are used to administer the user access control by each column family having separate access privilege level options, making column family the basic unit of access control. Each column family must be created and specified before data can be recorded under any column key the column family incorporates. Bigtable architecture compresses stored data by grouping data by the column families, requiring columns inside families to house the data of similar type. [47]

Versioning in Bigtable is implemented through timestamping multiple versions of cell data. Different cell versions are indexed by a 64-bit integer timestamp and sorted in a descending order to retrieve the latest version more quickly. Cleaning the expired cell versions is implemented per column family basis that can be configured to keep only a number of the latest versions or delete versions older than set time. A timestamp can be either assigned by Bigtable to be the time of the recording at microsecond accuracy, or the timestamps can be supplied from the client with a constraint of unicity. [47]

Figure 9 (based on [47]) demonstrates an example of a sensor metadata, which stores the sensor coordinates and its status. The row key is the sensor ID sensorID_43927 and it has data in two column families: location and status. The location column family has two columns latitude and longitude denoting the corresponding world coordinates, while the status column family has only one column thus not requiring a qualifier. The status column has three versions of the data at timestamps t5, t5, and t8, while the other fields have no versioning.
5.2.2. **Infrastructure and Architecture**

A Bigtable cluster generally runs on a shared machine pool, often overlapping processes from other services, and is reliant of Google’s other technologies. Bigtable operates on Google File System (GFS) and uses it to store log and data files. The data is stored in SSTable file format that provides a “persistent, ordered immutable map from keys to values”, lookup operations for value retrieval by a key, and iterating over a range of key-value pairs. [47]

Resource locking is performed by Chubby, a “highly-available and persistent lock service” utilizing the Paxos algorithm [47]. Chubby is used by Bigtable for various tasks [47]:

- master server management ensuring there is only one master at a time,
- discovering and finalizing the death of tablet servers,
- storing column family information (schema) for the Bigtable instance,
- storing access control lists, and
- storing bootstrap information location for Bigtable data.

The master server is the first of the three major components of the Bigtable implementation. The master server keeps track of the pool of available tablet servers and manages the server tablet machines by assigning tablets, handling column family schema alterations, detecting changes in the tablet server cluster, balancing load on the tablet server cluster, and handling garbage collection in the file system. The master server is not under a heavy load due to clients communicating directly with the tablet servers thus not requiring data to pass through the master server. The tablet server, being the second of major components, handles a number of tablets and performs read and write operations and also divides tablets growing too large into smaller tablets. The third of the major components, the client library, manages and directs requests to appropriate tablet servers and returns the responses back to the client application. [47]

The data replication is performed at the GFS level and the tablet servers hold only one copy of the data available in the Bigtable level, potentially leading to short periods of unavailability of data in case of a node failure, due to latency it takes to reload the data from the GFS to a new tablet server [47]. Bigtable architecture does not compromise data consistency and elects to forgo availability instead, placing Bigtable to CP (consistency, partition tolerance) category in CAP model and PC/EC (partition, consistency, else, consistency) in the extended PACELC model [30].

![Bigtable data model illustration of sensor location and status.](image)

Figure 9. Bigtable data model illustration of sensor location and status.
5.2.3. Google App Engine

The Bigtable platform is made available to the consumers through the Google App Engine (GAE) system. Google App Engine is a Platform-as-a-Service (PaaS) type cloud computing platform for developing and deploying web applications on Google infrastructure clusters. The web application can be served from the user’s web domain using Google Apps suite or from a provided free domain under.appspot.com. GAE supports runtime environments and software development kits (SDK) for Java technologies, Go (Google’s programming language), and Python versions 2.5 and 2.7 for application programming. GAE also features a local development environment, application versioning, and direct application deployment. [48]

GAE automatically scales the web application to meet the number of requests by virtualizing the application across multiple machines in a secure sandbox. The sandbox restricts the application’s possibilities to communicate with other computers over the Internet to the provided URL fetch and email services. The application can be connected only through HTTP (Hypertext Transfer Protocol) or HTTPS (Hypertext Transfer Protocol Secure) protocol requests on standard ports. GAE enforces additional time constraints on the application, such as requiring a query or a task to receive a response from the application within 60 seconds. Web applications on GAE are also restricted from writing on the file system and are only allowed to read static files that were uploaded together with the application code. Writing on the file system is carried through one of Google’s data storage services, such as Memcache, Cloud SQL, Cloud Storage or App Engine Datastore. [48]

The App Engine Datastore is the Bigtable data store instance available for consumers. In addition to choosing whether or not to enforce the data schema, the developer can designate additional columns to be indexed, though not all data types are indexable. The Datastore can be queried through Google Visualization API Query Language (GQL), an SQL-like query language. GQL enables querying the Datastore with SQL syntax with some limitations. Inequality filters are limited to one property, leading to some of the more complex queries, such as standard geospatial boxing, not being supported at the database level. Other common functions in SQL that are not supported include functions like aggregate functions and JOINs. Datastore service also provides ACID transactions using optimistic concurrency control, which assumes that transactions complete without conflicting each other, thus not requiring resource locking. For larger data blobs, the Cloud Storage service provides object storage for files and objects up to terabytes in size. [48]

5.3. Amazon Dynamo

Amazon.com, one of the world’s largest e-commerce operators servicing tens of millions of customers at peak times, faces the challenges of service reliability in massive scale. Services implemented on top of a server infrastructure of tens of thousands of machines are continuously prone to both small and large scale hardware component failures in data centers located around the globe. [36]

Amazon’s Dynamo is key-value data store architecture that focuses on massive scaling while providing high availability and performance even under heavy network partitioning and server failures. Some of Amazon’s core services requiring high
availability run on Dynamo infrastructure. Amazon’s governing approach to the system modeling is service efficiency: services and clients are placed under service level agreements (SLA) that enforce additional quality constraints such as latency requirements and expected request rate distribution. Amazon’s SLA requirements are generally expressed in 99.9th percentile of the distribution. [36]

Figure 10 (based on [36]) illustrates an abstraction of Amazon’s decentralized infrastructure focusing on supporting web services. The dynamic content delivered to the client or a service by the page rendering components may consist of data from many other services. Some of the requested content may be processed by aggregator services creating composite responses from gathering and combining intermediate data from several other services or data stores. [36]

5.3.1. Data Model

Dynamo is a collection of key-value pairs that are indexed and retrieved by the primary key, providing only simple read and write operations to the data item uniquely identified by the key. The lightweight nature of the data structure and operations help key-value stores gaining performance, but it comes with the cost of the more advanced features such as complex querying. Due to key-value stores tying all data manipulation strictly to the keys, the rest of the data is opaque to the query. In comparison to other types of stores, the simplicity of the data model grants higher performance, but at the same time prevents any operations to non-key fields. Similarly, due to the database operations’ single-key restriction, operations spanning multiple items are not supported. The schemaless data store is designed for

![Figure 10. Abstraction of Amazon’s service platform infrastructure.](image)
applications and services with relative small data item sizes, usually less than 1 Mi.
[36]

5.3.2. Infrastructure and Architecture

Dynamo utilizes a variant of the consistent hashing technique to data partitioning and replication. It also provides access to incremental scalability: the cluster can be expanded one machine at a time. Load is distributed across storage hosts based on the hash function output range, forming a wrapping, fixed circular hash value space. Nodes in the cluster are denoted by an assigned random value to position them in the hash ring. The nodes house the data items, identified by a key, that are distributed to the hash ring based on the key’s hash value. [36]

The variant of the consistent hashing used by Dynamo addresses some of the issues in the basic version, namely the random position assignment and node heterogeneity handling. The load imbalance caused by the random assignments and heterogeneous host performances are alleviated through the creation of virtual nodes, which appear as normal nodes to the system and are often denoted as “tokens” in the architecture. A node can be responsible for multiple virtual nodes, effectively occupying several positions in the hash ring space, further balancing the load fluctuations in cases of node failures and additions. The virtual node concept also addresses the node heterogeneity issue by allowing more powerful nodes to operate additional virtual nodes. [36]

Data replication is performed through designated coordinator nodes replicating the data items in its range to a number of following distinct physical nodes, moving in clockwise direction in the hash ring [36]. The nodes holding a particular key are listed in the preference list, and the system design ensures that all nodes are able to determine the nodes in the list for any key [36].

Figure 11 (adapting [36, 49]) illustrates how the hash keys (white circles) are divided between nodes (blue dots) and how the nodes are replicated to three next neighbors in the consistent hash ring. For example, node F contains all the hash keys between the previous node (B) and its own location (data: C, D, E, F), while the node F replicates itself to the next three nodes, advancing clockwise, i.e. the replica of node F can be found stored in nodes I, K, and L.

The simple circular architecture enables adding and removing nodes with relative ease due to actions only affecting the immediate neighbor nodes. New nodes are inserted through a membership management system working on a gossip-based propagation protocol. Failing nodes are observed through passive detection, where a node is marked as failing if it fails to respond to the other nodes’ requests. The failure information is propagated through decentralized gossip-based failure detection protocols. Service requests towards a failed node are directed to the alternate nodes while the failed node is periodically pinged to determine whether it has become alive again. This kind of passive, distributed detection system avoids having to maintain a centralized node information registry and also preserves node symmetry. [36]
Dynamo places in AP (availability, partition tolerance) category in CAP theorem. During normal conditions, Dynamo loosens data consistency to attain lower operational latencies, and in the case of partition occurring, Dynamo gives up data consistency to ensure availability, placing Dynamo in PA/EL (partition, availability, else, latency) in PACELC model [30].

Data consistency is handled through the eventual consistency method (Chapter 4.2.), data versioning, and a quorum-like protocol that requires a minimum number of nodes to participate in read and write operations to be successful. Eventual consistency propagates the changes asynchronously, eventually making all replicas up-to-date. Each modification is stored as a new immutable version of the data and the latest version usually supersedes the older versions. Network partitions may cause data version branching, which has to be processed according to data causality to determine if the data versions are in conflict, or to be handled by application side reconciliation. Data causality is examined through vector clocks, which are effectively lists of (node, counter) tuples associated with every object and version. If vector clocks are unable to determine causal ordering of versions on parallel branches, the version reconciliation has to be performed on the application side. [36]

5.3.3. Amazon DynamoDB

Amazon’s older NoSQL database service, SimpleDB document store, offered flexible query capabilities, but at the same time traded off scalability and
performance in favor of full indexing. Severe scalability limits, such as hard limiting the request capacity and storage capacity, a 10 Gi domain (account) hard limit, made SimpleDB unfit for many applications requiring higher performance and capability. [50]

Amazon DynamoDB was released in January 2012, after many years of in-house development [51]. Based on the Amazon’s Dynamo architecture while taking the best parts of Amazon’s SimpleDB architecture, DynamoDB offers fully managed, schemaless, cloud-based, and highly scaling NoSQL database service that offers customers a service free of all administrative tasks such as hardware setup, configuration, partitioning, and replication [50, 51]. Instead of query flexibility, DynamoDB focuses on achieving the highest performance and reliability [50]. DynamoDB integrates with other Amazon Web Service (AWS) products, such as Identity and Access Management (IAM), Simple Storage Service (S3), CloudWatch, and several others [50].

To further emphasize the performance focus, DynamoDB runs on solid state drives (SSD) to achieve the highest performance and the lowest latencies with disk operations. SSDs also enhance I/O performance in comparison to traditional hard disk drives (HDD). Downside of the high performance SSD storage is the increased storage costs due to SSD media being more expensive in comparison to HDD, although DynamoDB somewhat compensates on this by SSDs enabling lower costs of handling requests. [50]

AWS provides DynamoDB SDKs for Java, .NET and PHP, while other AWS SDKs, such as Android, iOS, and Ruby, also support DynamoDB [50, 52]. Some third party libraries, such as boto for Python, exist to provide an interface to AWS services, including DynamoDB.

DynamoDB offers two kinds of primary keys: a hash type primary key, and a hash and range type primary key. A hash key consists of one attribute in an unordered hash index. The hash key attribute must be a unique identifier in the table hash key index, and can be either of a string, an integer, a float, or a binary type. A hash and range type key consists of two attributes forming a unique composite key. A hash primary key attribute is constructed in the same manner as in the single key attribute option, but it no longer has the requirement of being unique in the table hash index. A number of range primary key attributes, which are built with a sorted range index, can share the same hash key attribute as long as the combination of the key attributes results in a unique pair. In similar fashion to the hash key attribute, a range key attribute can be a string, an integer, a float, or a binary type value.

Based on the selected primary key type, an item consists of one or two key attributes that are comprised from a name and a value pair. In addition, a data item may contain an unlimited number of attributes, each consisting of a name-value pair, only to be limited by the 64 KiB item size restriction. Figure 12 illustrates the DynamoDB data structure hierarchy. [50, 52]
DynamoDB operates the database through hash key attributes. Any and all query operations are limited to a single hash key, but DynamoDB offers also a hash and range key option to create a composite key. After accessing the hash key, comparative operations are available to be applied to the sorted indexed range key attribute. Other item attributes are normally opaque to the query, and thus unavailable as parameters, but DynamoDB recently (April 2013) introduced a feature to create up to five local secondary indices (LSI) per table. Unlike their name suggests, however, the local secondary indices are not true secondary indices in SQL sense, but more like additional range keys to query the table, keeping the limitation of querying only a single hash key still in effect. The local secondary index tables are automatically created and maintained after setting the secondary indices, but they also impose a potential scalability issue: each individual hash key in the table is limited to a 10 Gi in total size of its items and their index entries. To query attributes across hash keys, secondary index tables must be designed, created, and maintained manually. [52]

Designed for relatively small item objects, DynamoDB limits data item size to 64 KiB, which includes attribute values and attribute field names. The number of attribute fields is not limited as long as the total size of the item stays below the item

---

**Figure 12. DynamoDB data structure hierarchy.**

ACCOUNT

TABLE

ITEM

**HASH KEY**

- Name
- Value

**RANGE KEY**

- Name
- Value


dots

TABLE

...
size limit. For larger data items, such as multimedia data, it is recommended that the data itself is stored in another storage service, such as Amazon Simple Storage Service (S3), and only a pointer or a link to the data object is stored in DynamoDB. The DynamoDB account and the tables have no size restrictions, although the number of tables is initially limited to 256, but more can be requested directly from Amazon through a request form. Query and scan operations are limited to returning a result set maximum of 1 Mi in size or, but if halted by limits, the query also returns ExclusiveStartKey attribute, which allows the next query to continue from the last key processed. [50, 52]

Amazon DynamoDB uses a provisional throughput model to handle the request throughput. The throughput capacity unit is specified as one read or write operation per second processing up to 4 KiB of data. Larger item operations require more capacity units to complete: 1 unit for every 4 KiBs, rounded up. Every table has provisional throughput specified on both reads and writes. Provisional throughput is soft capped at 10 000 per table, but can be increased over 10 000 per table and over 20 000 account wide by contacting Amazon through a request form. The provisional throughput allocation is basically the only thing left to be administrated by the user or administrator. Throughput utilization can be monitored through CloudWatch service, which also allows placing alarms, which send an e-mail when a set percentage of the total capacity is being used. [50, 52]

DynamoDB can be administrated, along other AWS products, through AWS Console service, or through API commands. Provisional throughput increases are limited to the maximum of 100% increase per call through API, but multiple calls can be made to increase the throughput further, unlike decreasing the throughput, which is limited to four times a day. AWS Console allows manual increasing of throughputs without the 100% step limit. The table throughput increases take effect in a few seconds or minutes while decreases can range from minutes to a few hours. [52]

5.4. Windows Azure Storage

Windows Azure is Microsoft’s cloud computing platform offering services for hosting and managing applications and web sites, computational capacity, cloud storage services, virtual machines, and several others services. In addition to a hosted relational Windows Azure SQL Database service, Windows Azure Storage offers three scalable storage services: Blob Storage for storing large, unstructured objects, Table Storage for storing schemaless entities, and Queue Storage for message delivery. [53]

Windows Azure Table Storage is Microsoft’s fully managed, fault-tolerant NoSQL key-value data store with high scalability and availability. Azure Table Storage provides a structured data store with relaxed data schema working on distributed and replicated file system layer. [53]
5.4.1. Data Model

Similar to DynamoDB, Azure Table Storage is a key-value store featuring a very simple data structure of indexed key-value pairs with the rest of the properties being opaque to the query. Azure Table Storage shares all the advantages and the limitations of DynamoDB’s data model, with one exception: Azure Table Storage has the ability to perform partial table scans due to range-based partitioning architecture [54]. This allows a range query to be performed on the partition key property equivalent to DynamoDB’s hash key attribute, but it has its downsides such as lower performance and higher costs due to scan operations performed on multiple partitions [54].

5.4.2. Infrastructure and Architecture

Figure 13 (based on [54]) illustrates a high level abstraction of the Azure Storage architecture, where VIP stands for Virtual IP and DNS for Dynamic Naming System. The basic unit of the Azure Storage architecture is the storage stamp, a cluster of storage node racks. Storage stamps typically consist of 10 – 20 racks and offer from a few to a few dozen petabytes of raw storage per stamp. Each node rack creates an isolated fault domain with redundant networking and power resources to further manage failure states. Additionally, the utilization level of storage stamps is kept under 80% to reserve capacity to handle rack failures. Data replication and migration is performed through inter-stamp replication. The storage stamps are handled by the location service, which also manages the account namespace across the stamps and allocates the accounts to storage stamps. [54]
A storage stamp consists of three fundamental layers. The front-end layer is a set of stateless servers handling the incoming requests, authentications, and request routing to partition servers. The front-end layer can also directly access large objects in the file system layer and cache the frequently accessed data. [54]

The partition layer manages the partitioning of data objects through object’s partition key, thus providing the scaling aspect of the database management system. Load balancing is managed on this layer through distributing the load across the partitions, each served by a single partition server. The partition layer is also responsible of resolving the storage data type abstractions (table, blob, or queue), data migrations to other storage stamps, and managing the asynchronous data replication across storage stamps (inter-stamp replication) to provide disaster recovery. [54]

The stream layer is the distributed and replicated file system which supports the scaling and durability of Windows Azure storage services. This layer stores the bits to the file system and manages the data replication across nodes within the storage stamp (intra-stamp replication). [54]

Azure Table Storage uses a range-based architecture, which utilizes dividing an internal Object Table data structure to continuous, non-overlapping ranges of rows that are distributed to partition servers based on traffic loads. RangePartition load balancing is implemented through three operations that dynamically manage the number and distribution of RangePartitions based on the partition loads. [54]

Load balance operation monitors the partition servers identifying and redistributing RangePartitions with high load. Split operation splits a single overloaded RangePartition into smaller RangePartitions and redistributes them across partition servers. Merge operation combines subsequent cold or lightly loaded RangePartitions within the Object Table to form a single RangePartition to prevent the total number of partitions from inflating due to split operations. [54]

Range-based partitioning does not scale well to sequential access key schemes such as timestamping, which executes all writes in the end of RangePartition. The balancing system is unable to effectively balance such access patterns, leading to a loss of performance. [54]

The stream layer provides database availability by replicating data across nodes, while the partition layer provides consistency guarantees [54]. The layered architecture allows decoupling the nodes providing strong consistency from the nodes that provide database availability, effectively allowing maintaining a strong consistency and providing availability in case of partition occurrence [54]. Additionally, Azure Storage targets specific fault scenarios, in which it can provide both availability and strong consistency [54]. This categorizes Azure Storage as PA/EC (partition, availability, else, consistency) in PACELC model.

### 5.4.3. Azure Table Storage Service

Azure Table Storage entities can be up to 1 Mi in size and have up to 256 properties with three of them being mandatory: a partition key, a row key and a timestamp. A partition key and a row key form the entity’s primary key used for querying. A partition key is used for partitioning entities in the same table while row keys, together with partition keys, uniquely identify the entity. The timestamp is an opaque
property set by the system and is used for optimistic concurrency control in cases of multiple write accessing. Tables are partitioned through partition keys, and all entities having the same partition key are grouped together in the same partition, leading to faster querying of row keys inside a partition. [55]

Azure Table Storage offers slightly more diverse key querying in comparison to Amazon’s DynamoDB. In addition to the range querying of the row key property, Azure Table Storage supports partial table scans for querying partition key ranges, enabled by the range-based partitioning architecture [54]. However, partial scanning in most cases is very slow due to scan reading several partitions residing on different partition nodes, and in worst cases, an inefficiently chosen partition key range may result in full table scan, leading to very slow processing and high cost [54]. Azure Table Storage does not support secondary indices, thus querying by other than key properties requires either a full table scan or manually built and maintained secondary index tables [55].

Storage services have a hard size limit of 200 Ti per account, although a single subscription can have up to five storage accounts by default [54]. In addition to the database account hard size limit, the number of transactions is limited to 2000 operations a second per partition and 20000 operations a second account wide, as well as account bandwidth limits ranging at 5 - 15 Gi per second, depending on traffic type and redundancy options, are hard limits with no possibility to increase them. [56]

Azure Storage maintains three replicas of the data locally to provide data durability. In addition, Azure Storage offers an option for geo-replication. The Geo Redundant Storage system maintains three additional replicas of the data in another datacenter in the same region as the primary storage, granting additional protection against major disasters [54].

Windows Azure offers software development kits (SDK) for .NET, Java, Node.js, PHP, and Python. The Windows Azure services can also be accessed through HTTP calls, REST API, or LINQ (Language Integrated Query) and ADO.NET services. [53, 55]
6. APPLICABILITY OF DATABASE SERVICE TO SENSOR DATA

In the scenario targeted in this thesis, the sensors produce data with a very simple data structure. Each sensor produces a timestamp and one or more values, which can be numeric or text strings. The cloud database service is chosen according to the following requirements: database scaling, absence of administration, data querying, and service pricing.

6.1. Technical Aspects

Amazon SimpleDB document store offers flexible querying provided by its full indexing, which enables the execution of more complex queries like geospatial boxing and placing conditional operators on more than one attribute. However, the severe built-in constraints to the domain (account) sizes and query frequencies place hard scalability limits, rendering SimpleDB unsuitable to fulfill the other requirements for the database implementation.

A Bigtable instance through Google App Engine (GAE) service platform (App Engine Datastore) has no such limitations to scalability as Amazon’s SimpleDB has. The GAE Datastore offers more limited query capabilities in comparison to SimpleDB document store, but still allows more flexible querying than key-value stores DynamoDB and Azure Table Storage provide. However, GAE is designed to accommodate hosted web applications, requiring an uploaded server side web application to communicate with the Datastore. This would require an additional communication step, and additional bandwidth and processing expenses, as the clients uploading the sensor measurement data cannot directly communicate with the database itself but have to reroute the data through the web application.

The decision needs to be made between the key-value stores of Amazon DynamoDB and Windows Azure Table Storage, which both have their respective pros and cons. Azure Table Storage offers slightly better querying through partial table scans, however the partial scans are slow and taking the sensor data structure into account, the partial scans would often be across many partitions. Partition key queries can also potentially result in very slow and expensive full table scans, if the parameters are incorrect. Azure Table Storage has also data size limit 200 Ti per storage account as well as transaction limits to up to 20000 operations a second per account and up to 2000 operations a second for partitions. The transaction numbers are performance target values for optimal cases and, as such, are advertised as ‘up to’ values.

To scale the database instance beyond the 200 Ti account size hard limit, manual sharding has to be performed to divide the data across multiple Azure storage accounts. 20 storage accounts are provided per subscription by default and more can be requested by contacting Microsoft customer support.
On the other hand, Amazon’s DynamoDB offers very high scalability, performance, and availability, but has higher storage costs due to SSD-based data storage. DynamoDB does not have any hard limitations to hinder scaling, unless using the local secondary index feature, which imposes a 10 Gi hard limit for each secondary indexed hash key, including index tables. Table 4 presents an overview of the features of the discussed cloud Database-as-a-Services.

Table 4. An overview of the features of the discussed cloud database-as-a-Services

<table>
<thead>
<tr>
<th>Database-as-a-Service</th>
<th>Category</th>
<th>Scalability</th>
<th>Availability</th>
<th>Querying</th>
<th>PACE/LC</th>
<th>External connection</th>
<th>Database size hard limit</th>
<th>Transaction hard limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Google AppEngine Datastore</td>
<td>Wide column</td>
<td>Yes</td>
<td>Yes</td>
<td>Variable indexing, inequality operations limited to one column</td>
<td>PC/EC</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Amazon SimpleDB</td>
<td>Document</td>
<td>No</td>
<td>Yes</td>
<td>Fully indexed and flexible</td>
<td>PA/EL</td>
<td>Yes</td>
<td>10 Gi / domain</td>
<td>25 / second</td>
</tr>
<tr>
<td>Amazon DynamoDB</td>
<td>Key-value</td>
<td>Yes</td>
<td>Yes</td>
<td>Single hash key and range keys</td>
<td>PA/EL</td>
<td>Yes</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Microsoft Windows Azure Table Storage</td>
<td>Key-value</td>
<td>Yes</td>
<td>Yes</td>
<td>Single partition key and row keys. Partial table scans</td>
<td>PA/EC</td>
<td>Yes</td>
<td>200 Ti / storage account</td>
<td>Up to 20000 / second</td>
</tr>
</tbody>
</table>

6.2. Service Pricing

Both of the remaining service providers, Amazon Web Services and Microsoft’s Windows Azure, feature a sufficient free tier capacity for their NoSQL database service, which is useful when developing the application. However, the service providers price their products by utilizing very different metrics, thus making direct pricing comparisons not immediately apparent for all the metrics. Table 5, Table 6, Table 7, and Table 8 present the pricing tables (April 2013) for Azure Table Storage and Amazon DynamoDB (Ireland region)¹.

¹ Windows Azure Table Storage: http://www.windowsazure.com/en-us/pricing/details/storage/
   AWS DynamoDB: http://aws.amazon.com/dynamodb/pricing/
Table 5. Azure Table Storage pricing

<table>
<thead>
<tr>
<th>Transactions</th>
<th>$0.01 / 100k /month</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transaction limit</td>
<td>Up to 20000 / 2000 / second</td>
</tr>
<tr>
<td>Storage</td>
<td>$0.095 - $0.037 /Gi / month</td>
</tr>
<tr>
<td>Data in</td>
<td>$0</td>
</tr>
<tr>
<td>Data out</td>
<td>$0.12 - $0.05 / Gi</td>
</tr>
<tr>
<td>Account size limit</td>
<td>200 Ti</td>
</tr>
</tbody>
</table>

*Account / Partition hard limit*  
*Depends on the total size of stored dataset and the chosen replication scheme (Table 7)*  
*All data transfer in is free.*  
*The first 5 Gi / month is free, after that the pricing follows Table 8*

Table 6. Amazon DynamoDB pricing

<table>
<thead>
<tr>
<th>Read capacity</th>
<th>$0.00735 / 50 / hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write capacity</td>
<td>$0.00735 / 10 / hour</td>
</tr>
<tr>
<td>Capacity limit</td>
<td>20000 / 10000 / second</td>
</tr>
<tr>
<td>Storage</td>
<td>$0.283 / Gi / month</td>
</tr>
<tr>
<td>Data in</td>
<td>$0</td>
</tr>
<tr>
<td>Data out</td>
<td>$0.12 - $0.05 / Gi</td>
</tr>
<tr>
<td>Account size limit</td>
<td>None</td>
</tr>
</tbody>
</table>

*Capacity reserved for tables*  
*Capacity reserved for tables*  
*Account / table per second, soft limited: a customer may contact Amazon for more capacity.*  
*All data transfer in is free.*  
*The first 1 Gi / month is free, after that the pricing follows Table 8*

Table 7. Windows Azure Storage data storage monthly pricing

<table>
<thead>
<tr>
<th>Storage capacity</th>
<th>Geo-redundant</th>
<th>Locally redundant</th>
</tr>
</thead>
<tbody>
<tr>
<td>First 1 Ti</td>
<td>$0.095 / Gi</td>
<td>$0.070 / Gi</td>
</tr>
<tr>
<td>Next 49 Ti</td>
<td>$0.08 / Gi</td>
<td>$0.065 / Gi</td>
</tr>
<tr>
<td>Next 450 Ti</td>
<td>$0.07 / Gi</td>
<td>$0.06 / Gi</td>
</tr>
<tr>
<td>Next 500 Ti</td>
<td>$0.065 / Gi</td>
<td>$0.055 / Gi</td>
</tr>
<tr>
<td>Next 4,000 Ti</td>
<td>$0.06 / Gi</td>
<td>$0.045 / Gi</td>
</tr>
<tr>
<td>Next 4,000 Ti</td>
<td>$0.055 / Gi</td>
<td>$0.037 / Gi</td>
</tr>
<tr>
<td>Over 9,000 Ti</td>
<td>Contact Microsoft</td>
<td></td>
</tr>
</tbody>
</table>

Table 8. Data transfer pricing tables for Windows Azure and AWS platforms

<table>
<thead>
<tr>
<th>Data transfer per month</th>
<th>Price per Gi per month</th>
</tr>
</thead>
<tbody>
<tr>
<td>Up to 1 Gi AWS /5 Gi Azure</td>
<td>Free</td>
</tr>
<tr>
<td>Up to 10 Ti</td>
<td>$0.120 / Gi</td>
</tr>
<tr>
<td>Next 40 Ti</td>
<td>$0.090 / Gi</td>
</tr>
<tr>
<td>Next 100 Ti</td>
<td>$0.070 / Gi</td>
</tr>
<tr>
<td>Next 350 Ti</td>
<td>$0.050 / Gi</td>
</tr>
<tr>
<td>More</td>
<td>Contact Amazon/Microsoft</td>
</tr>
</tbody>
</table>
The transactions performed are charged very differently due to DynamoDB’s provisional throughput system. Storage space costs are where Azure Table Storage comes ahead due to DynamoDB’s SSD based storage system. Neither service charges for data transfers between different services within the region.

In addition, both providers offer fixed-term contracts to gain discounts on bandwidth fees. Amazon offers a 1-year or a 3-year “Reserved Capacity” offering savings up to 53% and 76% respectively on capacity units. The contracts require an upfront payment and commitments to a minimum of 5000 read and write capacity units for the duration of the contract. Microsoft offers a 6-month and a 12-month plan offering a storage and transaction discount of 20-32% depending on monthly monetary commitment with either monthly or up-front payment options available.

### 6.3. Use Case Expenditure Analysis

The use case consists of 100 sensors, each storing a 10-digit numeric value in the data table and the data index table at 10 second intervals (0.1 Hz) for the duration of 3 years. Assuming the operations are distributed evenly, both the data table and the data index table are subjected to 10 requests a second. Approximating the data overhead caused by the data item naming scheme and indexing, the size of a single stored measurement value in the database can be estimated to be 86 bytes, index entry included. The transactions are less than 4 KiB in size, thus only one capacity unit per operation is consumed by DynamoDB.

At the pace of 10 requests a second for both tables, each table handles 25.92 million transactions a month, resulting in the grand total of 1.89216 billion transactions during the 3-year period. The total amount of data stored in the database monthly is 2.077 Gi, resulting in the final database size of 75.775 Gi after 3 years. Table 9 and Table 10 dismantle the itemization comprising the total expenditure of the use case according to the price information presented in Chapter 6.2., without taking into account the free tiers or fixed-term contract discounts. Azure Table Storage uses the geo-replication option.

Table 9. Azure Table Storage expenditure itemization generated by the use case

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Transactions</td>
<td>$5.184 / month</td>
</tr>
<tr>
<td>Data transfer</td>
<td>$0.036 / month</td>
</tr>
<tr>
<td>Storage after 3 years</td>
<td>$7.199 / month</td>
</tr>
<tr>
<td>Storage average</td>
<td>$3.599 / month</td>
</tr>
<tr>
<td>Total after 3 years</td>
<td>$12.419 / month</td>
</tr>
<tr>
<td>Total average</td>
<td>$8.819 / month</td>
</tr>
<tr>
<td><strong>Total cost after 3 years</strong></td>
<td><strong>$317.495</strong></td>
</tr>
</tbody>
</table>
The total costs after three years are in favor of Azure Table Storage, which incurs a 58.6% less expenditures in comparison to DynamoDB. Azure Table Storage generates the majority of costs of the use case from the transaction fees (58.8%) and storage costs (40.8%). Forfeiting the geo-replication safeguard would attain around 10% savings in the total costs after 3 years. DynamoDB’s transactions fees, presented by the provisioned write capacity units form 49.4%, and the storage costs form 50.2% of the total expenditures.

Windows Azure offers a maximum of 32% discount on the platform services when subscribing for a 12-month fixed-term contract. The fixed-term monthly commitments start at $500 a month, but to obtain the highest discount, a commitment of a minimum of $40 000 monthly spend across the platform is required, and paid in advance. In addition, the contract discount cannot be combined with the graduated discount of the data storage or the data transfer presented in Table 6; the base tier price is used instead.

Unlike the fixed-term contracts provided by Windows Azure platform, the Amazon DynamoDB Reserved Capacity fixed-term contracts are DynamoDB specific. The 3-year Reserved Capacity contract offers 76% discount on provisional throughput capacity expenses. The minimum amount of Reserved Capacity is 5000 read units and 5000 write units, which totals in a $12 204 (May 2013) upfront payment for EU (Ireland) region. The Reserved Capacity contracts do not affect any other billing aspect.

Table 11 and Table 12 present the discounted use case expenditures using the maximum discount rates provided by the fixed-term contracts. The total expenses of Azure Table Storage are 32% cheaper in comparison to the non-discounted price, but the ratio degrades if the system is using the discounted pricing tiers for the data storage or the data transfer (Table 7 and Table 8). In comparison, the discounted price for DynamoDB is 37.7% cheaper than the non-discounted price, but the use case does not utilize the full capacity of the fixed-term contract as it does not take into consideration the minimum of 5000 units of reserved capacity for read operations.
Table 1. A discounted use case expenditure itemization for Azure Table Storage

<table>
<thead>
<tr>
<th>Item</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transactions</td>
<td>$3.525 / month</td>
</tr>
<tr>
<td>Data transfer</td>
<td>$0.024 / month</td>
</tr>
<tr>
<td>Storage at 3 years</td>
<td>$4.895 / month</td>
</tr>
<tr>
<td>Storage average</td>
<td>$2.448 / month</td>
</tr>
<tr>
<td>Total at 3 years</td>
<td>$8.445 / month</td>
</tr>
<tr>
<td>Total average</td>
<td>$5.997 / month</td>
</tr>
<tr>
<td><strong>Total cost after 3 years</strong></td>
<td><strong>$215.897</strong></td>
</tr>
</tbody>
</table>

Table 2. A discounted use case expenditure itemization for DynamoDB

<table>
<thead>
<tr>
<th>Item</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write capacity</td>
<td>$2.540 / month</td>
</tr>
<tr>
<td>Data transfer</td>
<td>$0.036 / month</td>
</tr>
<tr>
<td>Storage at 3 years</td>
<td>$21.444 / month</td>
</tr>
<tr>
<td>Storage average</td>
<td>$10.722 / month</td>
</tr>
<tr>
<td>Total at 3 years</td>
<td>$24.021 / month</td>
</tr>
<tr>
<td>Total average</td>
<td>$13.298 / month</td>
</tr>
<tr>
<td><strong>Total cost after 3 years</strong></td>
<td><strong>$478.740</strong></td>
</tr>
</tbody>
</table>

6.4. The Decision

The decision rests between the highest performance and scalability of Amazon’s DynamoDB and the lower operating and storage costs of Microsoft’s Windows Azure Table Storage. The difference in bandwidth scalability is further depicted by the transaction ‘up to’ hard limit target of Azure Table Storage and the off-the-shelf bandwidth offered by DynamoDB. The maximum amount of transactions provided by Azure Table Storage is up to 20000 requests a second in optimal conditions, while DynamoDB offers the double of that, 20000 read operations and 20000 write operations a second, before having to contact Amazon for more capacity. The transaction scalability limits have a serious impact due to the final application potentially scaling to magnitudes of sensors constantly producing data. Taking into account DynamoDB’s performance, approachability, and secondary indices, as well as Azure Table Storage’s potentially limiting factors, such as storage size limits per storage account and transaction restrictions, the decision for database service is in favor of Amazon’s DynamoDB.
7. DESIGN AND IMPLEMENTATION

The challenges for design and implementation stem from the requirement of diverse, SQL-like query options coupled together with the high scalability and performance of a NoSQL system. In the case of key-value stores, these two requirements somewhat contradict with each other as one of the database aspects that the key-value type NoSQL systems sacrifice to achieve faster processing, lower response times, and better scaling, is query versatility. The requirements also create structural design problems originating directly from the philosophical view of relational database management systems, where data is in a normalized form and can be retrieved and formatted as long as the query statements are tweaked to accommodate the needs. This is in contrast to NoSQL databases, where the data is structured and denormalized (in SQL terms) in order to accommodate the pre-established queries.

This chapter presents the decisions behind the database design and covers some of the challenges and trade-offs required to create an efficient, working database solution. The query logic is described and application scenario queries are demonstrated. The structure and implementation of the database client used to access the database is presented with a testing phase validating the queries.

7.1. Data and Operations

The general structure of the database system is quite simple. The database should store metadata for each sensor including sensor ID, sensor type, world coordinates, altitude, and possibly other metadata defined by users. The sensors generate data in a form of a timestamp and one or more measured values. The sensors may produce data at variable intervals and the data may be integers, doubles or text strings. All sensor metadata should be able to be queried in several different ways. In addition to fetching the sensor information directly through a sensor ID, listing all sensors of a given sensor type is required. Also, the sensors must be available for querying through geospatial bounding box querying of world coordinates.

Querying the stored sensor data items should be able to be done through both measurement timestamps and the recorded values. Timestamp query requires supporting a timeframe search as well as returning the latest values, while data queries require returning data points according to inequality operations on data values.

7.2. Database Structure

Figure 14 demonstrates the first design of the database structure, featuring all the necessary attributes for storing sensor metadata and sensor data points as well as the manually built index tables enabling querying on non-key attributes. The SensorMeta Index Tables container illustrates the index tables used to provide query capabilities
to SensorMeta table attributes and the Sensor Data Tables container highlights the tables used for storing and indexing the sensor measurement data.

![SensorMeta and Sensor Data Tables](image)

Figure 14. The first database implementation.

The designed database scales to accommodate the vast amount of data and traffic produced by a wide network of sensors, and works around the potentially limiting aspects of DynamoDB, such as query indexing and item size restrictions, providing the required query capabilities. In addition, the unstructured data model enables a relaxed data schema, allowing users to define additional metadata attributes to complement the enforced metadata schema. These attributes, however, are unqueryable directly, but will be retrieved when sensor metadata is requested through sensor IDs.

In this chapter, the database design is described and the justification for the database structure is provided.

### 7.2.1. Selecting Primary Keys

The way that DynamoDB implements data keys is very simple. A hash type key consists of one attribute in an unordered hash index. The hash key attribute must be a unique identifier in the table hash key index, and can be either of a string, a number, or a binary type. Alternatively, a hash and a range type key may be selected instead. This type of key consists of two attributes forming a composite key: a hash key and a range key, which is indexed in a sorted index. The composite key formed must be unique within the table indexing.
The importance of key selection stems from the hash based partitioning architecture: the key attributes strictly define the available queries. A query can access only a single hash key and the set of range keys sharing the hash key value. This directly translates into a very severe limitation of range querying being confined to a subset of items presented by a single hash attribute. All other data attributes are opaque to the query apart from the key attributes, which makes performing queries on other attributes more complicated.

Any queries spanning over multiple hash keys must perform a full table scan. A scan operation enables for the results to be filtered quite extensively, but the scan operation reads the whole table in the memory before processing it, making scan very slow and expensive operation that should be avoided at all costs [52].

*SensorID* is a natural unique key for sensor metadata. Every sensor writing data in the same table would eventually bring issues with data management, such as data migration and user access, so the data is divided to tables by *SensorID*, with timestamp being a unique identifier within the sensor’s data points. Figure 15 illustrates the basic data tables containing the sensor metadata and measurement data.

![Figure 15. The basic data tables containing sensor metadata and measurement data.](image)

However, the querying provided by the table indexing only allows to get information for a single sensor based on *SensorID* and get a single value from the data table based on a timestamp. In addition, the model presented in Figure 15 has another aspect to consider: expenditures. Since DynamoDB uses a provisional throughput system with minimum reserved capacities of one read and write operation a second, using the current data table division scheme would accumulate significant expense overhead due to great majority of sensors operating at frequencies far below 1 Hz. Splitting *SensorID* into a composite ID value consisting of *GroupID* and *SensorID* allows sensor groups of arbitrary size to be created. This reduces the reserved capacity overhead as well as provides methods of fine tuning group-based user access management. Grouping the data tables also requires storing *SensorID* in the data table hash key attribute to identify the set of values stored by a specific sensor. The timestamp is moved to the range key attribute, enabling to query the timestamp with inequality operations, thus providing queries based on time frames. Figure 16 presents the new table structure implementing the ID scheme change.
This structure stores all the required attributes, but provides very limited query variation. Efficient query operations accessing non-key attributes are not supported due to the system architectural decisions, and any operations targeting the non-key attributes require a full table scan. The requested query operations must be implemented through other means to satisfy the application specifications.

In addition, this table naming scheme, coupled with DynamoDB table name character limitations, create some limitations to the possible GroupID value scheme. DynamoDB allows only English letters [A-Z, a-z], numbers [0-9], and three special characters, which are underscore, dash and period [_, -, .] to be used in the table names. These characters define the allowed GroupID scheme apart from the period character, which is used to separate the GroupID value and the index table order number, so it is restricted from the GroupID value scheme.

### 7.2.2. Building Index Tables

To work around the architectural limitations in order to achieve better query versatility, data denormalization can be utilized to manually build secondary index tables to provide additional indices to query on. Building secondary index tables manually means redundantly copying (denormalizing) the data to an additional table, which stores the data in a fashion that enables queries to be performed on the desired attributes. Increased storage expenditures and more complex implementation are the prices to pay to enable more flexible querying.

To be able to query sensor’s non-key metadata attributes across all sensor IDs, a secondary index tables storing appropriately structured data must be created. The TypeIndex table stores Type as a hash key attribute, while combining GroupID and SensorID in the range key attribute to satisfy the requirement of the uniqueness of a key. The TypeIndex table allows retrieving all sensor IDs that have a common Type value. The current implementation only returns the sensor IDs. To be able to return other metadata apart from the sensor IDs from a TypeIndex query, the additional metadata should be copied over to the TypeIndex table.

Similarly, the GroupIndex table is created to enable listing of existing sensor GroupIDs. Without a separate table providing an index for the GroupIDs, the only method of retrieving them would be to perform a full table scan on the SensorMeta table.

However, sensor groups do not have a common parameter to divide them, so a common dummy value must be used as a hash key attribute instead to enable querying over GroupIDs stored as the range key attribute. GroupIDs are unique
within the group ID set, so the combination of a dummy hash key and a GroupID satisfy the key uniqueness condition. Using a common dummy hash key has a potential scalability issue due to architectural data partitioning being based on the hash key attribute, although it is unlikely if the sensors are grouped in a sensible manner. In case of the amount of sensor groups out-scaling the partition capacity, a data division scheme must be implemented. Cross-division queries cannot be performed without multiple queries and additional client side processing.

LocationIndex table must be built to enable metadata querying by sensor coordinate values. The key selection for the LocationIndex table is a bit more challenging task due to bounding box nature of geospatial queries. DynamoDB features no methods to perform inequality operations on multiple attributes, which becomes a problem when querying sensor locations within bounding boxes.

Using the coordinate axes, latitude, longitude, and altitude, as hash key attributes with the coordinate values in the range key attributes would provide simple and easy querying for a coordinate value, but the composite key formed does not uniquely identify sensors i.e. many sensors may have the same coordinate value in the given axis. This leads to each individual coordinate value being presented by a single index table item, which must contain all the desired sensor information for every sensor present in the given coordinate. In other words, the single index table item must contain sensor IDs of all the sensors present in the particular coordinate slice. Due to highly scaling nature of the application, this presents a potential scalability issue resulting from the 64 KiB item size limit of DynamoDB.

To solve the scalability issue, the sensor information stored in the index table must be distributed to avoid the item size restriction. This can be achieved by appending the sensor IDs to the range key attribute coordinate value. Appending the unique Sensor IDs to the range key attribute forms unique range key attributes, resulting in each sensor having its own individual coordinate index entries for each coordinate axis. A hash (‘#’) character is used to separate the coordinate values from the attached sensor ID values. Figure 17 illustrates the index tables used for querying the non-key sensor metadata attributes.

<table>
<thead>
<tr>
<th>TypeIndex</th>
<th>GroupIndex</th>
<th>LocationIndex</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;hash&gt; Type</td>
<td>&lt;hash&gt; Dummy</td>
<td>&lt;hash&gt; Axis</td>
</tr>
<tr>
<td>&lt;range&gt; GroupID</td>
<td>&lt;range&gt; GroupID</td>
<td>&lt;range&gt; Coordinate</td>
</tr>
<tr>
<td>#SensorID</td>
<td></td>
<td>#GroupID #SensorID</td>
</tr>
</tbody>
</table>

Figure 17. Index tables for SensorMeta table querying.

Due to DynamoDB architecture not supporting comparisons between multiple attributes, performing bounding box queries with an altitude component on the LocationIndex table illustrated in Figure 17 would require three queries returning the whole data set satisfying the inequality conditions, one for each axis. Afterwards, the client software must process the retrieved data sets to find the sensor IDs satisfying all three coordinate axis conditions and provide the resulting data set to the user.

This kind of querying comes with a potentially heavy cost in traffic and processing. If the conditions are very broad, returning large sets of data items for each component may result in high expenditures. It is also possible to create queries that retrieve large datasets for every condition, but eventually displaying an empty result to the user due to conditions having no common items, eliminating all of the returned results.
The implementation of the location search presented in Figure 17 is very inefficient and potentially expensive, thus further consideration is required to improve the performance of the query. The availability of querying each coordinate axis individually is not really required, which allows transforming the index table structure to perform queries based on a single axis and storing other coordinate information in non-key attributes. In addition, the sensor Type attribute can also be stored to make queries based on both sensor location and sensor type more efficient.

Due to a single hash key only remaining in the LocationIndex table, it basically functions as a dummy hash key grouping the whole data set under it. With a latitude value and sensor IDs stored in the range key attribute, a query will result in a single data set storing information for every coordinate axis returned based solely on the latitude axis conditions. This data set is further filtered according to the longitude and altitude axis conditions, resulting in roughly one third of processing required in comparison to the previous implementation. Figure 18 presents the improved LocationIndex table.

<table>
<thead>
<tr>
<th>LocationIndex</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;hash&gt; Dummy</td>
</tr>
<tr>
<td>&lt;range&gt; Latitude#GroupId#SensorId</td>
</tr>
<tr>
<td>Longitude</td>
</tr>
<tr>
<td>Altitude</td>
</tr>
<tr>
<td>Type</td>
</tr>
</tbody>
</table>

Figure 18. Improved LocationIndex table.

Similarly to GroupID, the sensor coordinates do not have a common nominator, requiring either to use a dummy value to enable range querying over every sensor’s coordinate values with a single query, or implementing a data partitioning scheme. The dummy hash key presents a potential scalability issue due to the architectural partitioning based on a hash key. A partitioning scheme, such as dividing locations according to a grid, would eliminate the potential scalability issue, but also prevent querying over all sensors with a single query.

Mobile sensors pose another problem. Storing the location as sensor data provides location history, but including the mobile sensors in the location searches of LocationIndex table cannot be done in an efficient manner. The LocationIndex is sorted by latitude but not by sensor IDs. Thus, finding the index entry for a certain sensor cannot be done without a full table scan. This makes LocationIndex more suited for static sensors that do not change their location. Each time a sensor changes its location, the client must first get the sensor’s last location from the SensorMeta table or the data table (or from application memory), and use the previous location to find and delete the previous location index entry from the LocationIndex table. Then the client must write the updated location information in the SensorMeta entry as well as create a new LocationIndex item for that sensor. Having a large amount of mobile sensors that update their location frequently would cause a significant amount of traffic due to several operations required for a single value update, thus querying mobile sensors through LocationIndex is not supported (Chapter 8.3. presents a potential solution for updating sensor location for mobile sensors).

Querying the stored data by the Value attribute requires building an index table with a data structure enabling query statements to process the Value attributes. The
Data tables are named dynamically through the GroupID's value, thus appending a suffix to distinguish the index tables is utilized. The 'base' data table (shown in Figure 16) will receive a suffix '.00' and a suffix '.01' is assigned to the first index table. The index table hash key attribute is SensorID as it will, together with the index table name, identify the data set stored by a specific sensor. Value is stored in the range key attribute to enable querying over the stored values of a given sensor.

However, the Value attribute is not a unique value, thus the SensorID-Value key scheme does not satisfy the uniqueness requirement of the composite key. This can be solved in the same way as with LocationIndex table by appending a unique attribute, in this case a timestamp, to the values, creating a unique range key attribute. The key structure allows querying the Value attributes of a given sensor with inequality operations and returning the values and their timestamps. If a sensor records more than one value, an additional index table must be created for each additional value generated by the sensor to enable the querying of those attributes. Figure 19 exhibits the data table and the two index tables for sensors recording two values, while Figure 20 assembles the whole database design with the auxiliary metadata attribute ValueType to store the sensor value format information (defined in Chapter 7.4.). Figure 20 also highlights the base data tables and the index tables to emphasize the additional measures required to implement the query capabilities.

![Figure 19. Data table storing two values and its index tables.](image)

![Figure 20. The assembled database design emphasizing table function distribution.](image)
DynamoDB stores the attribute name together with the attribute value, which may lead to unexpectedly large overhead expenses especially in data tables and their index tables. Truncating attribute names of *GroupID*, *SensorID*, and *Timestamp* to *GID*, *SID*, and *TS* respectively saves 24 bytes per data item, 12 bytes in the data table and 12 bytes in the index table, in comparison to the design in Figure 20. This might not seem like a significant amount at first, but for example the sensor setup of the use case example in Chapter 6.3. (100 sensors with data rate of 0.1 Hz for 3 years) would produce roughly 21.15 Gi less data in the three years’ time if each data item would require 24 bytes less to store. Applying the attribute naming scheme alteration, the first complete database implementation is presented in Figure 14.

**7.3. Performing the Operations**

This chapter presents the challenges in the querying resulting from the chosen key structure of the index tables. The composite range key attributes of the *LocationIndex* table and the data table index tables require additional attention due to the numeric values being transformed into character strings, which directly affects how the query comparison is performed. This chapter also presents the solutions to handling the lexical query comparisons.

The forced inclusion of the sensor IDs into the range key has one major consequence: a numeric value is no longer a number; it is now a character string. This makes querying the values different from a numeric comparison, and more complicated. The string comparison works through lexical principles, in other words the comparison takes the first character of each string and compares their UTF-8 (Universal Character Set Transformation Format, 8 bit) order value. If the character UTF-8 order values are equal, the next characters’ order values are compared until the decision is made.

Figure 21 demonstrates a lexical string comparison example giving an invalid response in contrast to a numeric comparison.

As a string comparison only measures the absolute value of the UTF-8 character order and does not take into account the potential negativity of the numeric value of the string, which leads to situations like querying “‘-14’ > ‘-12’” returning *TRUE*. For example a character string version of integer range of [-99, 99] requires 2 digits and orders from smallest to largest as following:

\[
\]
The value disorder in comparison to numeric values causes all the inequality operations for negative numbers to be altered. Using the above value set, querying for “values less than ‘-50’” requires transforming the query to “values between ‘-50’ and ‘-’”, where ‘-50’ is the lower bound and ‘-’ is the upper bound, the colon character (‘:’) having an UTF-8 order next from ‘9’, thus ‘-’ covering all negative values but still being “smaller than ‘0’”. A query between positive and negative values is not supported due to such a query requiring two distinct queries and aggregating the resulting data sets. For example “values between ‘-1’ and ‘1’” would be transformed to one query for “values less or equal to ‘-1’” and another query for “values between ‘0’ and ‘1’”. This limitation could be hidden from user with extra work on the client. The only queries returning both positive and negative results are “less than” and “less or equal” operations on a positive value, and even in this case the negative values are returned in a reverse order.

The additional complexity caused by the lexical sorting requires performing some additional formatting to the location query to be able to get all the proper values due to the attached sensor IDs. For example, querying for values “equal to ‘12’” from the LocationIndex table will return an empty result due to the absence of a pure ‘12’ string stored in the database. Instead, all strings are followed by attached IDs, making the data format into ‘12#GroupID#sensorID’.

Due to the appended sensor IDs, retrieving all valid records for the example query of “equal to ‘12’” requires transforming the query into a query of values “between ‘12’ and ‘12$’”. The dollar character’s (“$”) order value is next from hash character’s (“#”) order value i.e. $>$ #, thus making the range cover value ‘12’ and all attached IDs. In case of strings of float numbers, a similar technique is used but a string of ‘000001’ is attached to the fraction part of the query instead. This gives the latitude coordinate value floats in the database a maximum precision of 6 fraction digits and the numeric sensor data values get a string of ‘000000000001’ appended to them, granting 12 digit precision to data values.

In addition, the discrepancy in the lexical comparison results (Figure 21) between variable length strings requires transforming and storing the values as fixed length and leading zero padded to produce the desired query comparison results. A comparison between zero padded character strings ‘019’ and ‘137’ produces a numerically correct result after the first character comparison operation \( \text{ord}(0) < \text{ord}(1) \). Storing additional ‘unnecessary’ characters generates costs, but it is required to be able to produce valid query results.

To solve the query challenges of strings of negative numeric values, a value mapping scheme can be utilized. The sensor coordinates use WGS-84 (Word Geodetic System) standard, placing the latitude value range to [-90, 90]. Knowing the range of possible values for the attribute allows mapping the values to a positive range. Performing a mapping operation of ‘+90’ to every latitude coordinate value sets the latitude range to [0, 180]. This way the client does not have to process the query input values ranging from negative to positive. Longitude and altitude values do not require value mapping due to being stored in numeric format and being processed client side.

The data tables and their index tables proved to be more challenging to be handled due to arbitrary data formats and data ranges produced by the sensors. Instead, an additional sensor metadata attribute containing the value format information must be stored in the SensorMeta table. The ValueType attribute is created by the client when the sensor is inserted to the database. The attribute stores the type of Value (integer,
double, or string) and the number of characters reserved for the integer part of a numeric value to manage the leading zeroes required by lexical queries.

Table 13, Table 14, Table 15, and Table 16 present how the sensor metadata of three sensors is stored in the SensorMeta table and its index tables. The SensorMeta table entries (Table 13) exhibit two user defined attributes: Owner and Project. Even though the example table displays an empty attribute field for the sensors that do not have the additional attribute defined, the attribute fields do not actually exist for those sensors. The latitude components of the range key attributes stored in LocationIndex table (Table 14) have been mapped to a positive value range.

Table 13. SensorMeta entries for the example data set

<table>
<thead>
<tr>
<th>&lt;hash&gt;</th>
<th>&lt;range&gt;</th>
<th>Type</th>
<th>Latitude</th>
<th>Longitude</th>
<th>Altitude</th>
<th>Owner</th>
<th>Project</th>
<th>Value Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test</td>
<td>Test01</td>
<td>Temperature</td>
<td>65.05002</td>
<td>25.46671</td>
<td>5.42</td>
<td>CSE</td>
<td>D2</td>
<td></td>
</tr>
<tr>
<td>Test</td>
<td>Test02</td>
<td>Temperature</td>
<td>65.16</td>
<td>24.81</td>
<td>2</td>
<td>68193722</td>
<td>I2</td>
<td></td>
</tr>
<tr>
<td>Test</td>
<td>Test03</td>
<td>Humidity</td>
<td>65.05002</td>
<td>25.46671</td>
<td>13</td>
<td>CSE</td>
<td>D2</td>
<td></td>
</tr>
</tbody>
</table>

Table 14. LocationIndex entries for the example data set

<table>
<thead>
<tr>
<th>&lt;hash&gt;</th>
<th>&lt;range&gt;</th>
<th>Latitude#GID#SID</th>
<th>Longitude</th>
<th>Altitude</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lat</td>
<td>155.05002#Test#Test01</td>
<td>25.46671</td>
<td>5.42</td>
<td>Temperature</td>
<td></td>
</tr>
<tr>
<td>Lat</td>
<td>155.05002#Test#Test03</td>
<td>25.46671</td>
<td>13</td>
<td>Humidity</td>
<td></td>
</tr>
<tr>
<td>Lat</td>
<td>155.16#Test#Test02</td>
<td>24.81</td>
<td>2</td>
<td>Temperature</td>
<td></td>
</tr>
</tbody>
</table>

Table 15. GroupIndex entries for the example data set

<table>
<thead>
<tr>
<th>&lt;hash&gt;</th>
<th>&lt;range&gt;</th>
<th>GID</th>
<th>Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dummy</td>
<td></td>
<td></td>
<td>Test</td>
</tr>
</tbody>
</table>

Table 16. TypeIndex entries for the example data set

<table>
<thead>
<tr>
<th>&lt;hash&gt;</th>
<th>&lt;range&gt;</th>
<th>GID#SID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Humidity</td>
<td>Test#Test03</td>
<td></td>
</tr>
<tr>
<td>Temperature</td>
<td>Test#Test01</td>
<td></td>
</tr>
<tr>
<td>Temperature</td>
<td>Test#Test02</td>
<td></td>
</tr>
</tbody>
</table>

Table 17 and Table 18 demonstrate an example of a small data set stored by the sensors presented in Table 13. Due to hash based indexing, the hash key attributes are grouped but not sorted, unlike the range key attributes, which are sorted indexed. The real value of the first item input for Test01 is ’24.10’, but DynamoDB automatically truncates trailing zeroes from numeric values.
Table 17. Data table Test.00 entries for the example data set

<table>
<thead>
<tr>
<th>&lt;hash&gt; SID</th>
<th>&lt;range&gt; TS</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test01</td>
<td>2013-04-11-09-10-00</td>
<td>24.1</td>
</tr>
<tr>
<td>Test01</td>
<td>2013-04-11-09-10-20</td>
<td>24.22</td>
</tr>
<tr>
<td>Test01</td>
<td>2013-04-11-09-10-40</td>
<td>24.29</td>
</tr>
<tr>
<td>Test01</td>
<td>2013-04-11-09-11-00</td>
<td>24.41</td>
</tr>
<tr>
<td>Test03</td>
<td>2013-04-11-09-10-00</td>
<td>64.8</td>
</tr>
<tr>
<td>Test03</td>
<td>2013-04-11-09-11-00</td>
<td>64.9</td>
</tr>
<tr>
<td>Test02</td>
<td>2013-04-11-09-10-00</td>
<td>26</td>
</tr>
<tr>
<td>Test02</td>
<td>2013-04-11-09-10-30</td>
<td>26</td>
</tr>
<tr>
<td>Test02</td>
<td>2013-04-11-09-11-00</td>
<td>27</td>
</tr>
</tbody>
</table>

Table 18. Data table Test.01 index entries for the example data set

<table>
<thead>
<tr>
<th>&lt;hash&gt; SID</th>
<th>&lt;range&gt; Value#TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test02</td>
<td>26#2013-04-11-09-10-00</td>
</tr>
<tr>
<td>Test02</td>
<td>26#2013-04-11-09-10-30</td>
</tr>
<tr>
<td>Test02</td>
<td>27#2013-04-11-09-11-00</td>
</tr>
<tr>
<td>Test03</td>
<td>64.8#2013-04-11-09-10-00</td>
</tr>
<tr>
<td>Test03</td>
<td>64.9#2013-04-11-09-11-00</td>
</tr>
<tr>
<td>Test01</td>
<td>24.10#2013-04-11-09-10-00</td>
</tr>
<tr>
<td>Test01</td>
<td>24.22#2013-04-11-09-10-20</td>
</tr>
<tr>
<td>Test01</td>
<td>24.29#2013-04-11-09-10-40</td>
</tr>
<tr>
<td>Test01</td>
<td>24.41#2013-04-11-09-11-00</td>
</tr>
</tbody>
</table>

Due to the limitation of a query to a single attribute makes it difficult to create more advanced searches, such as query operations on multiple attribute values, without additional secondary index planning and maintaining and multiple queries processed client side. For example, a scenario where an application requires the most recent values of Temperature type sensors in an area defined by the bounding box [65.0, 24.0; 66.0, 26.0], using the example data presented in Tables 13 – 18, utilizes both planned indices and processing of multiple query results.

The application starts by mapping the latitude value parameters to the positive range and querying LocationIndex table (Table 14) with the latitude range [155.0, 156.0], retrieving the data set shown in Table 19. The data set is then filtered programatically client side according to longitude, altitude, and type condition parameters. All returned items satisfy the longitude parameters, but the sensor type condition eliminates the Test03 sensor from the result set, and the client decouples the sensor ID information and maps the latitude value back to [-90, 90] range (Table 20). Without the Type attribute stored in the LocationIndex table, the application would have to perform another query to the TypeIndex table (Table 16) to retrieve the IDs for all Temperature type sensors and cross-reference the IDs satisfying the Type condition with the retrieved data set.
Table 19. Items retrieved from LocationIndex according to scenario parameters

<table>
<thead>
<tr>
<th>Dummy</th>
<th>GID#SID</th>
<th>Longitude</th>
<th>Altitude</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lat</td>
<td>155.05002/Test/Test01</td>
<td>25.46671</td>
<td>5.42</td>
<td>Temperature</td>
</tr>
<tr>
<td>Lat</td>
<td>155.05002/Test/Test03</td>
<td>25.46671</td>
<td>13</td>
<td>Humidity</td>
</tr>
<tr>
<td>Lat</td>
<td>155.16/Test/Test02</td>
<td>24.81</td>
<td>2</td>
<td>Temperature</td>
</tr>
</tbody>
</table>

Table 20. The result set filtered according to longitude and type parameters

<table>
<thead>
<tr>
<th>GID#SID</th>
<th>Latitude</th>
<th>Longitude</th>
<th>Altitude</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test/Test01</td>
<td>65.05002</td>
<td>25.46671</td>
<td>5.42</td>
<td>Temperature</td>
</tr>
<tr>
<td>Test/Test02</td>
<td>65.16</td>
<td>24.81</td>
<td>2</td>
<td>Temperature</td>
</tr>
</tbody>
</table>

Now the application has the sensor IDs of sensors fulfilling the location and type requirements. For each returned sensor, the application then queries the timestamp indexed data table. In this case, both of the queries target the Test.00 (GroupID) data table (Table 17), but one queries the hash key attribute Test01 (SensorID) and another queries Test02. The query is performed with an empty range key condition parameter to list all data entries, using additional query parameters to reverse the index traversing (i.e. start reading from the last index) and limit the amount of returned results to one. This type of query returns a single item that had the last index position, in this case, the largest timestamp. The result of the scenario process is two independent data items, presented in Table 21 although the items do not belong to the same data set unless aggregated by the application.

Table 21. The resulting items of the scenario

<table>
<thead>
<tr>
<th>SID</th>
<th>TS</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test01</td>
<td>2013-04-11-09-11-00</td>
<td>24.41</td>
</tr>
<tr>
<td>Test02</td>
<td>2013-04-11-09-11-00</td>
<td>27</td>
</tr>
</tbody>
</table>

Since DynamoDB uses the provisional throughput for tables, it is left to the system administrator to monitor and adjust the tables’ allocated read and write capacities in cases where new batches of sensors are inserted into the network system. In the case of exceeding the provisional throughput, boto library retries an operation 10 times before returning an exception. Amazon CloudWatch offers a service for monitoring and setting alarms to follow the load on the tables [52]. The SensorMeta table and its indices are not expected to require much administration, unless applications generate a heavy load of range queries, such as constant geospatial boxing.

Due to the 64 KiB item size limit, DynamoDB is not suitable for storing larger items, such as images. In these cases, the larger data blobs should be stored using other storage services, such as Amazon Simple Storage Service (S3), and store a pointer or a link to the data object in the DynamoDB data tables.
7.4. Access Management and User Accounts

AWS Identity and Access Management (IAM) service, a part of Amazon’s service cloud, allows securely creating and managing AWS user groups and permission policies. Each user or user group can be provided with private AWS access keys to connect to the DynamoDB instance. These permissions can be administered through the AWS Management Console or API calls. The AWS console sign-in link for IAM users is provided in IAM dashboard and can also be found in a csv (comma separated values) file when the user account and a random password are created. Accessing the AWS console in other than administrative purposes should not be required. AWS IAM service does not incur any additional costs.

A user group is created by simply naming the group and giving it resource access policies from a list of templates, which include options ranging for full AWS account access, apart from account and billing information, to a read only access to an individual AWS service. Access policies can be fine-tuned through Amazon’s access policy language, and additional constraints, such as access time frames or originating IP (internet protocol) addresses can be defined. Access policies can also be attached to the user instead of user group, should it be required. It is strongly recommended that the users are separated into user groups instead of everyone using a single master access key. The access key and the secret access key are stored to config.ini file (Appendix 1), which is read by the client to access the DynamoDB instance.

7.5. Client Implementation

The database client is programmed with Python 2.7 and uses boto 2.8.0 library\(^2\) for Amazon Web Services interface. The client consists of two distinct parts: One using graphical interface featuring database search and queries as well as adding and initializing new sensors to database system, and other, which is used to upload data from the sensors into the database. Figure 22 illustrates the general structure of the database client program, while Figure 23 presents the database client sequence diagram.

\(^2\) https://github.com/boto/boto
Figure 22. The general structure of the database client program.

Figure 23. Client sequence diagram.
The main function of the program is in databaseClient. It first calls for initialization of graphical user interface (GUI) from ClientGUI class and then starts the threaded authentication process by calling DatabaseAuthentication class. Finally ClientGUI class is called again to draw the main frames and turn over the control to GUI.

Due to connecting and changing regions take a little while to complete, these operations are performed in a threaded function. DatabaseAuthentication class is a threaded class that handles reading the config.ini file and retrieving AWS access keys and region information from the file. The AWS access information is sent to DatabaseLogic class, which initializes the actual connection to the DynamoDB instance. The DynamoDB instance handle is then sent to ClientGUI class. The config.ini file holds access keys and region settings used to connect to the DynamoDB account. The structure of the config.ini file can be found in Appendix 1. Currently, the database application resides on the Europe region (eu-west-1) servers that are physically located in Ireland [52].

The DatabaseLogic class creates the interface to AWS DynamoDB by utilizing boto 2.8.0 library. DatabaseLogic initializes the database connection and returns the instance handle. DatabaseLogic receives query parameters from ClientGUI and handles all the communication with the DynamoDB instance, returning query results back to ClientGUI for graphical presentation.

The ClientGUI class handles most of the workload in the application. The class builds the graphical user interface by utilizing Tkinter module, python’s built-in GUI programming toolkit. First, ClientGUI initializes the root frame that holds all other GUI elements inside it and draws the main level navigation menu in the upper part of the frame. The main level buttons function as menu selectors and they control the view of the lower area of the root frame. These views are ‘Search for sensors’, ‘Search sensor data’ and ‘Add new sensor’. Each of these views opens a new search frame that offers parameters and options for querying the selected task. The ClientGUI class consists of functions of graphical drawing of elements, display logic to decide what is shown, and user input validation and input reformatting, before sending the query parameters to DatabaseLogic for the actual querying.

‘Search for sensors’ view offers getting sensor metadata by sensor IDs, querying sensors by type, and querying the location by coordinates. Location can be queried by supplying parameters for latitude and longitude to geospatially box an area, with the option of also supplying an altitude parameter. ID and type queries only return exact matches, but location search allows for conditional operators to be utilized.

‘Search sensor data’ requires a valid GroupID and SensorID to be supplied to open the data query view. This is due to the client requiring sensor’s metadata to build the view. Querying for the latest data values checks the maximum of 100 items with reverse index progression, returning items starting with the largest timestamp value. For querying by data values, the user must select the index table to query from. This basically selects which value to query in the case of sensor recording multiple values. These index table numbers correspond to the order of values saved to the SensorMeta table.

‘Add new sensor’ view asks the user to supply GroupID, SensorID, sensor type, location information and the number of measured values for the new sensor. Also, due to string type queries, the user is asked to specify the amount of integer digits the numeric sensor recordings will use. Each extra digit costs roughly extra 1.9 Mi to be stored for each one million rows, half on data table and half on index table. Users can
specify additional sensor metadata attributes to be stored to the SensorMeta table. These, however, are unqueryable directly, but will be retrieved when sensor metadata is requested through sensor IDs. The software dynamically creates data tables for the new sensor according to a GroupID based schema. The new tables are initially allocated with 1 unit of read and write capacity. The provided throughput values can be increased through the AWS Console service or through API calls.

The other part of the client application is the dataUpload client. It is a simple UDP (User Datagram Protocol) server providing an application interface to third parties producing the sensor data. The application reads the AWS credentials from config.ini just like the main client and gets the metadata from the database. The UDP server waits for a string message, which consists of GroupID, SensorID, Timestamp and Values. The application checks sensor IDs and data formats against the metadata and then uploads the measurement data to the data table and the relevant index tables. The sensor ID check is to ascertain that the sensor metadata exist.

7.6. Testing

A prototype for the database and client implementation was built. The queries and functionalities presented in Chapters 7.1., 7.2., and the scenario in Chapter 7.3. were tested and established to be working.

The SensorMeta table provides queries based on a GroupID attribute. The user can retrieve the metadata of every sensor in a group, or get the metadata of a single sensor. Inequality operations are available to be applied to a SensorID attribute, but they are not implemented in the first version. Similarly, the GroupIndex table enables listing of all existing sensor groups through a dummy hash key attribute, and also allows querying GroupIDs with inequality operators. A BEGINS_WITH condition operation is implemented.

The TypeIndex table allows retrieving all sensor IDs that share the sensor type queried. The LocationIndex table returns a set of items based on a mapped latitude value. The retrieved item set can then be further filtered according to longitude, altitude, and sensor type parameters.

The <GroupID>.00 data table enables querying a SensorID (hash key attribute) of a sensor group (table name). The ‘.00’ suffix provides inequality operations to be performed on the timestamp attribute, allowing to query time frames. The returned data set includes the timestamps of the data points and their values. Similarly, the <GroupID>.01 table provides (the first) value indexing for the <GroupID>.00 data table. It enables querying for the (first) value attribute with inequality operations and returns the values and their timestamps. Additional values are indexed in index tables with an incrementing number suffix.
8. DISCUSSION

8.1. Conclusions

This thesis took only a slice from the NoSQL database management system pool, namely the large, proprietary systems offered as a hosted and managed database-as-a-service. A majority of NoSQL database management systems reside in the open-source domain. Not relying on proprietary, managed systems brings many popular NoSQL systems, such as MongoDB document store and Cassandra wide column store, available for deployment to fulfill various requirements for the database management system. The diversity of NoSQL systems results in various database management systems being accessed through standard API calls or SQL-like API query languages, such as Google Visualization API Query Language, instead of wide-spread standard like SQL for relative database management systems.

The NoSQL database management systems provide performance and scalability way beyond what the typical relational database management systems might offer, but they often require trade-offs in some areas, ranging from query versatility to consistency issues, often necessitating additional processing on the client side.

The resulting database design provides high scalability and performance required by the application scenario. In addition, it delivers the required basic query functionality through extensive use of manually constructed index tables, although the more complex queries, such as composite queries and multi-value queries, require additional client side processing, which needs to be implemented on case-by-case basis.

One of the issues in the development process was not to apply the “common knowledge” of the relational SQL world. The NoSQL systems work very differently and focus on different aspects, sometimes resulting in even some of the most basic features of SQL DBMS non-existent when dealing with the NoSQL scheme, and especially with the key-value stores. This became obvious through the rough initial designs being discarded several times due to attempting to implement designs sound by the relational standards, only to realize that the design simply will not allow querying the values intended.

The most challenging aspect to cope with was the strict limitation of only allowing queries to target only a single hash key without the possibility to span across several hash keys. From the SQL standpoint, not being able to query the primary key index is quite hard to fathom and to adapt to. Similarly, manually building and managing the secondary index tables to enable required queries through redundant data is something not often utilized with the relational database management systems, due to data denormalization working counter to the typical RDBMS design normalization process.

Unlike the traditional relational databases, the data in NoSQL databases is structured to accommodate the predefined queries instead of formalizing the data structure. The available queries of the database are strictly defined by the indexing decisions, thus making the indices dependent on both the structure of the data and the
queries required. In addition, the potentially high cost of storing and updating an additional secondary index table makes careful index planning imperative.

Also, the relative immaturity of the NoSQL system scheme made the development more time consuming as the tools and the systems themselves are still evolving. New features and operations were implemented quite frequently to both the database services and the programming library, often affecting the structural decisions of the database design.

8.2. Future Work

One possibility to be considered in the future could be a hybrid database system. In the hybrid system, the sensor metadata and its querying could be implemented in an SQL system, or some other system featuring flexible queries, enabling more diverse and complex querying of sensor metadata, while the data would be stored into DynamoDB tables to retain the speed and scaling of a NoSQL database system. Since DynamoDB does not use data relations, replacing the metadata portion of the database system does not break the database architecture. Relevant changes must be made to the client application to abstract the differences in the underlying architectures. Figure 24 illustrates the hybrid system for sensor metadata and sensor data.

![Figure 24. Hybrid database for sensor metadata and sensor data.](image)

The provisional throughput can be utilized most effectively when sensor writings on database are evenly distributed. However, if a data source has several write operations to be performed at the same time to the same sensor group, a batch write operation should be implemented and used instead. Using batch write instead a set of individual writes conserves provisional throughput capacity and helps to alleviate
potential write operation spikes. Also the fixed-term contracts offered by Amazon can yield significant discounts to the provisional throughput expenses.

Some kind of data partitioning scheme should be implemented to prevent cold, stale data residing on an expensive high performance storage system. Old data could be migrated to Amazon’s Simple Storage Service (S3). One option could be a yearly partitioning scheme with data tables, where each data table consists of a single year.

For more exhaustive data mining, reporting and analysis, it might be worth to investigate the capabilities of Amazon RedShift and Elastic MapReduce (EMR) services. Redshift provides advanced business intelligence capabilities for complex data analysis queries with SQL-based interface. Amazon EMR can process vast amounts of data by using distributed virtual servers running in Amazon cloud [57]. Amazon EMR uses Apache Hadoop framework together with an SQL-based engine Hive to provide a powerful toolbox for Big Data analysis and management with SQL-like queries [57]. However, Amazon’s other services are outside the scope of this thesis.

8.3. Local Secondary Indices

DynamoDB recently (April 2013) implemented local secondary index (LSI) feature, which could, in some use cases, enable non-key attribute querying without having to manually maintain a secondary index table. Local secondary index acts as an additional range key attribute for the table, and as such has the same limitations as the regular range key attribute. Any query spanning across multiple hash keys still has to be implemented through manual index tables or full table scan.

LSI places additional constraints to be taken into account with sensor grouping. The LSI group is defined by the attribute name and all items in the group must have the same data type (number, string, or binary), requiring all the sensors in the indexed group to store only one data type. LSIs are defined when the tables are created and incorporating additional LSIs later cannot be done. LSIs are sparse indices, meaning that if an item does not have the indexed attribute, it will not appear in the index table. This makes it possible for example to define two LSIs, but have some of the sensors in the group storing only one value without incurring additional costs or empty items from the indexing scheme.

In addition, LSI sets a hard size limit of 10 Gi per hash key in the table with LSI applied. The 10 Gi consist of all the items having the hash key as well as the LSI entries for hash key’s every LSI attribute the table has defined.

The LSI feature would be useful when applied to the data table indexing. LSI enables querying the numeric data table values directly without having to deal with the lexical string comparisons, handling leading zeroes, modifying query parameters, or storing auxiliary data format information in the SensorMeta table. Mapping the latitude coordinates in LocationIndex table is a simple task due to the fixed value range, but arbitrary value ranges and value types of sensor measurement values make implementing a value mapping scheme more complicated. The LSI feature would make both querying and data insertion a simpler task due to additional input formatting being no longer required. The lack of order discrepancy between positive and negative values would also enable searches ranging from negative values to
positive values to be performed with a single query without mapping the values to a positive range.

Figure 25 presents the data table design for a sensor generating two data values utilizing the LSI feature for data table indexing. The data table \(<\text{GroupID}>\) consists of two LSI groups denoted by the \(<\text{LSI}>\) tag: one for \(\text{Value1}\) and other for \(\text{Value2}\). LSI also slightly contributes to storage cost savings due to not having to incorporate additional index information to the index table structure. Even though the data index tables are not visible in the design, the index tables still exist, but are maintained by DynamoDB instead.

<table>
<thead>
<tr>
<th>(&lt;\text{GroupID}&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;\text{hash}&gt;) SID</td>
</tr>
<tr>
<td>(&lt;\text{range}&gt;) TS</td>
</tr>
<tr>
<td>(&lt;\text{LSI}&gt;) Value1</td>
</tr>
<tr>
<td>(&lt;\text{LSI}&gt;) Value2</td>
</tr>
</tbody>
</table>

Figure 25. A data table indexed with LSI.

However, one of the problems of relatively young technologies is that new functionalities may be implemented that are not yet fully supported by the environment technologies. In the case of LSI, this feature was published to the DynamoDB service API near the ending phase of this development project, and is also not yet fully supported by the library used in the client development.

In future implementations, utilizing LSI for data table secondary indexing instead of manual index tables would be beneficial. This would however necessitate a data partitioning scheme to be implemented in order to avoid the potential scalability issue of 10 Gi size limit per secondary indexed key imposed by LSI.

### 8.4. Improving Location Queries

Implementing the support for location queries for mobile sensors would have been very inefficient due to the multitude of operations required to perform a single location update to the index. Utilizing the LSI feature the \(\text{LocationIndex}\) table can be transformed to provide indexing by both sensor IDs and latitude. Figure 26 presents the table structure allowing finding a sensor through the sensor IDs stored in the range key attribute, as well as querying the latitude value through a LSI table.

<table>
<thead>
<tr>
<th>LocationIndex</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;\text{hash}&gt;) Dummy</td>
</tr>
<tr>
<td>(&lt;\text{range}&gt;) GID#SID</td>
</tr>
<tr>
<td>(&lt;\text{LSI}&gt;) Latitude</td>
</tr>
<tr>
<td>(\text{Longitude})</td>
</tr>
<tr>
<td>(\text{Altitude})</td>
</tr>
<tr>
<td>(\text{Type})</td>
</tr>
</tbody>
</table>

Figure 26. \(\text{LocationIndex}\) design supporting mobile sensors.
However, this method works efficiently only when using a dummy hash key to group all items into a single partition. If the data is partitioned by applying a variable hash key attribute, the situation reverts back to client first having to retrieve the previous location before processing the location index item. This works counter to the 10 Gi total size limit per hash key attribute set by LSI. In this case, the LSI must include every attribute to the LSI table, effectively limiting the size of LocationIndex table to 5 Gi without a data partitioning scheme.

In addition, LSI implemented on the latitude attribute of SensorMeta table would enable location queries, similar to those provided by LocationIndex, to consist of only a single sensor group (GroupID).

Another potential technology to be considered is Geohash\(^3\) mapping the coordinate values. Geohash is a hierarchical, grid-based geocode system that maps the world coordinates into a variable length encoded string, presenting a grid unit at precision set by the length of the geocode [58]. Geohash would provide encoding latitude and longitude values to a single index attribute by interleaving the values in a bitwise fashion, making the queries more efficient, as well as granting a natural hash key for data partitioning purposes. Due to Geohash gradually increasing precision when traversing the encode string, a string prefix works as a hash key, containing all the coordinates inside the grid element defined by the prefix.

The idea behind Geohash is the point proximity search based on common prefixes [58]. However, edge case locations of the Geohash bounding box grid may result in situations where the points are spatially close to each other, but share a very short common prefix [58]. This has to be taken into account in queries by processing the neighboring grid elements of the target point.

Utilizing Geohash would require complex processing of the query parameters on the client side. In addition, Geohash queries are complex due to grid-based architecture requiring processing the neighboring grid elements to include all potential results and filtering the resulting set with algorithms to remove the data points not part of the query.

The GeoHash queries result in a set of small queries in comparison to the one potentially large query of the first database implementation. With smaller data sets, the first implementation is more efficient due to Geohashing generating additional expenditures with multiple queries, but Geohashing may be worth of consideration, if the data set grows large enough to warrant implementing data partitioning schemes.

\(^3\)http://geohash.org
9. SUMMARY

This Master’s thesis presents the theory, design and implementation of a scalable cloud database solution for a sensor network, which is a part of the project called Internet of Things. To avoid having to manage server cluster infrastructure, a fully managed, cloud-based database-as-a-service (DaaS) platform is acquired from one of the major cloud service providers. The requirements for the DaaS also include high database scalability and performance processing the vast datasets generated by the magnitudes of sensors, as well as the ability to execute queries on the stored data.

NoSQL database management systems are designed to handle vast amounts of data, also called Big Data, through distributed architectures utilizing redundancy to achieve scalability, availability, and partition tolerance. NoSQL systems have to sacrifice features such as complex querying or strong data consistency to achieve the scaling properties often required by the Web 2.0 applications.

The three large cloud service providers, namely Google, Amazon, and Microsoft, were compared to select the most suitable DaaS. Google’s App Engine Datastore is restricted to web applications and does not allow external connections, thus it does not support the local clients uploading the data. Amazon’s SimpleDB provides flexible querying, but it is not designed for scalability. Microsoft Windows Azure Table Storage provides reasonable features at reasonable price, but has potential scalability issues with hard account size and transaction limits. Amazon DynamoDB, is designed for the highest performance and scalability without limiting factors, leading it to be selected as a target system.

The challenge was to create a database design that enables the required query types with the severely limited query functionality. This was achieved through denormalization process by manually creating secondary index tables to provide indices for the queries, allowing query flexibility in exchange for increased storage and transaction costs. Still, not all required queries are possible to be performed efficiently. To be able to perform such queries efficiently on the database side, a hybrid database system could be considered. In addition, Amazon Web Services provide several other services that can support DynamoDB in areas such as access management, service monitoring, and data mining.

Amazon DynamoDB fulfills the scalability requirements set by the application and manages to perform the required queries efficiently, apart from the more advanced multi-value condition queries targeting several attributes and tables.
10. REFERENCES


11. APPENDICES

Appendix 1  Example Contents of config.ini File.
Appendix 1 Example Contents of config.ini File

```
[CREDSENTIALS]
aws_access_key_id = BLSWI2NKV1P7LUUFOQHF
aws_secret_access_key = VsIse49L+Unxe9nNJDzAM/TEs9OzxLQmoZu1Wbq

[REGION_SETTINGS]
aws_service_region = eu-west-1
```