No-SQL Systems
DynamoDB, Couchbase, REDIS

Group 04
Damian Franco, Trey Sampson

May 6, 2022

1 Introduction

Due to the ever-growing and changing world of big data, we decided to take a look at some of the options for storing and organizing the massive amount of data collected by social media companies, online marketplaces, and other organizations in the technology industry. Not only is the size of data continuing to grow at an exponential rate, but the file types and access requirements are being updated, database protocols are being created and refined, and more traditional SQL databases are having trouble keeping up in some cases. SQL databases have been the industry standard for many years, but some of the changes in data organization (or lack thereof) have created issues with how SQL databases handle data. This is where *No-SQL* databases enter the conversation, as they can expand on the abilities of SQL databases without having to adhere to the fairly rigid SQL syntax. No-SQL databases are databases that store data in a format other than strictly relational SQL tables. Some argue that No-SQL stands for "Non-SQL" while others say it stands for "Not only SQL," but the important thing is that the data structures allow for more flexibility and power than strictly-SQL databases.

Our goal for this paper is to research three major players in the No-SQL database market. There are too many database options to cover in the time allowed, so we chose three commonly used programs that demonstrate different implementations of No-SQL. The databases that we seek to compare are DynamoDB, Couchbase, and REDIS. Of the three, DynamoDB is owned by Amazon, while the latter two have open-source licenses. For each of these database solutions, we would like to take a look at how the data is modeled and managed, how the database performs under certain conditions, how each database can be implemented, the structures of queries in each database, and what the best use cases are for each.

In section 2, we will talk about *DynamoDB* and how it give an overview of the various aspects of that No-SQL system. We will then set our eyes on discussing the dynamics of *Couchbase* in section 2 and *REDIS* in section 3. Lastly, we will draw some conclusions about these three No-SQL systems and other No-SQL systems in section 4.

2 DynamoDB

DynamoDB is a No-SQL database and cache service that supports key-value and document-based data structures. It was originally made to manage *Amazon.com*. *Amazon* promotes it as fully managed and serverless database. DynamoDB is only offered through *Amazon Web Services* (*AWS*). *AWS* is an online based platform that provides on-demand API's and cloud computing platforms for anybody willing to pay for the services. Some use cases of DynamoDB include individuals and companies who develop software applications in need of high user concurrency, scaling throughput, entertainment, retail operations and even gaming platform data management. *Amazon* also claims that DynamoDB has a very reliable performance even as it scales which will be an important topic when we talk about how well DynamoDB performs under different circumstances. Scaling is a very important topic to all No-SQL databases and is also emphasized for DynamoDB as well. We will expand more about how DynamoDB operates and the various characteristic of this No-SQL database.

Within the next subsections, we will cover how data is modeled within DynamoDB, how data is managed, the performance of DynamoDB, how queries are implemented in this No-SQL database, how users can implement it and the best use cases for this approach.

2.1 Data Modeling

Let's first discuss how data is modeled with DynamoDB. This database uses a key-value data structure which guides the modeling of the data. In key-value databases, data is denormalized. Denormalization (not to get confused with not normalized) of data allows for increased performance of a database, but with a draw back to how well write operations perform. This action also implies the basis of key-value No-SQL databases. On the contrary, normalized data design uses tables to store data. Data is placed into table or relation based on their similarities. Many SQL and relational model-based databases use normalized data structures. Instead of separating data and basing different attributes and descriptors in multiple tables, normalized data chooses to put all data in a single space where the user and machine could access all data. Machines will be able to access data through the unique key-value that was assigned to the object-like structure. This example of denormalized data is precisely what is implemented within DynamoDB.

Denormalized data is very important to how to implement a data model within DynamoDB. DynamoDB offers three main structures that are used for data modeling, tables, items and attributes. Starting at the lowest level, *attributes* are bits of data used to identify characteristics of an item. Attributes include identifies such as "*Name*" or "*Date*". An *item* is the central unit of controlling data. Items are similar to rows in a relational model and could be thought of by as an object in any object-oriented programming language. Lastly, *tables* consist of multiple items. Each item is identified by a *primary key* which uniquely identifies every item. This primary key is represents the key part of a key-value structure. There are two types of primary keys: a *simple* and a *composite* primary key. Simple primary keys only have one unique value that references and item. Composite primary keys consist of two parts: a *partition* and a *sort* key. The partition key is used to segment and distribute items into various segments used for sharding. We will speak more on sharding within the data management section. The second part of the composite primary key is the sort key which is used as an extra layer of ordering items and allows for deeper queries. With the use of these three main structures, the user can now model their data.

Primary keys are very important and should be understood very efficiently for any user that wants to implement a DynamoDB database. In fact, data modeling is a crucial step within any database implementation [1]. Having the knowledge of all dynamics of the basic modeling concepts will help users read and write to their application more efficiently. Integrity will continue to hold up well even when if errors occur on the back-end when a data model is written well. Some developers with experience within a relational database model would not have a tough time learning how to implement a DynamoDB model due to the small similarities between both models. Data management relies on the implementation of a correct and efficiently data model which we will discuss in the next section. Overall, DynamoDB contains a fairly straight forward data model and structures.

2.2 Data Management

Management of the data within DynamoDB will all be handled through the various services that *AWS* offers. Data backups, storage, and scaling of the database is very important to understand regardless of most of the management being automatically handled. Management of data can ensure high performance and be cost effective, but could also do the opposite by causing large cost amounts if the developer does not properly understand how they work.

Let us first discuss how DynamoDB is able to maintain high performance and availability within the data. DynamoDB scales horizontally by expanding a single table over multiple nodes. This database approach chooses to favor *availability* over consistency per the CAP theorem. Maintaining available data and scalability are DynamoDB's goals as per *Amazon* [2]. We also spoke about the importance of the partition key (also known as the "hash" key) within the previous chapter. The main use for the partition key is to help the machine access data to initiate a technique called consistent hashing. Hash is first generated by a hashing sequence that *AWS* handles and each partition key is used to retrieve data to hash. This technique will dynamically *partition* and distribute the data over a set of physical nodes (storage hosts) in the system. Each of the nodes in the system is set in a *hash ring*. Hash generated from the item is then sent to a node within the "hash ring" that is responsible for keeping that items data in storage. This approach of consistent hashing can cause an uneven distribution of data within the nodes which DynamoDB fixes with the use of virtual nodes added to the hash ring. These virtual nodes are added to an individual physical node where data can be "partitioned" correctly and evenly within the virtual node even if there is an uneven load distribution. This encompasses how partitioning is implemented to mange data.

Replication is the next important step in the managing of data that DynamoDB handles. Replication is the process of continuously storing the same data in multiple locations. This allows for the an improvement in availability which is the main ACID proprieties that DynamoDB focuses one and enforces little to no data loss within the database. We spoke about physical nodes of storage in the previous paragraph which is also is where replication occurs. Each item's data is replicated in every single physical node except for one of them. The node that is not replicated in the considered the "leader" node. This node is only responsible for the read and write operations and the distribution of the data that needs to be replicated to every other physical node. That covers the concept of replication within DynamoDB.

Amazon offers a service called "auto scaling" which allows for dynamically adjusting provisions throughput capacity through a fully managed experience. Auto scaling is important to note because

this features allows for real-time, dynamic scaling. Applications that have high user activity will also have valleys along with the high peaks of activity. Auto scaling enforces the idea that at any scale, there will always be consistent and good performance. We will talk more about how well auto scaling actually performs.

With the three proprieties above, DynamoDB is considered to have a distributed file system. *Replication*, *auto scaling* and *partitioning* have contributed to the strong performance of DynamoDB and little cases with data loss. Both proprieties being implemented well also allows for more improved enhancements of data management such as data versioning, scalability, and high performing read and write operations. There are some downsides to this way of data management. We will talk more about how this approach can have some performance cons within the next section. Overall, this section talks about three of the most important factors of data management within DynamoDB. There are many other algorithms and techniques that *Amazon* has implemented to ensure high performance, but most relate back to the three main topics that were spoken about within this section. All scaling, data storage operations, partitioning and replication is all handled by *AWS* which is why *Amazon* emphasizes that using DynamoDB is fully-managed.

2.3 Performance

One very important topic that consistently appears when trying to decide which No-SQL database to use is how well they perform under specific circumstances. Many times, users do not have the same implementation of a database. How many users are actively using the application, the amount of data being processed, data security and many other factors contribute to how well a database operates. Within DynamoDB, we have talked about its data modeling and management, so let's now take a look on how well they were implemented and how they perform in different circumstances.

We are going to first list the advantages that DynamoDB offers. We spoke in the previous section about how the auto scaling feature helps improve performance at any scale whether that being large or small data sets. That statement is true. With *Amazon's* auto scaling feature, many of databases have been able to perform at a high rate regardless of the size of the data sets. Auto scaling provides constant updates to match the the current data load which is shown to reduce costs by 30.8% compared to other No-SQL databases. Replication and partitioning are also very useful in preventing data loss and cutting costs. All reading, writing operations and storage of data are mostly cost effective within most cases. These are the main advantages that DynamoDB provides.

Unfortunately there are disadvantages and situations that arise when using DynamoDB. The first con that many people point out that is very expensive pricing when accessing functions within *AWS*. If a user has a very high data growth rate, then that could possibly break the bank for a user to maintaining high upkeep of their database. *Amazon* charges money for everything and performance is just one aspect that could get expensive quick. Moving on to more concrete details of disadvantages, low latency reads within the database take a bit of time to process. Processing low latency reads at a high rate within DynamoDB proves to be an issue which makes DynamoDB a not-so-great choice for a caching option. In fact, most consistent read and write operations within DynamoDB have been proven to fail due to the strong focus on availability rather than consistency.

2.4 Implementation

Amazon offers implementation through *Amazon Web Services* (*AWS*). Tables, their keys, and their attributes will all be implemented by the user through *AWS SDK*. Modeling of the data will have the user identify all characteristics of their data such as size, shape, velocity, sort order and related data. After the user implements all the code to handle all the data, an application to collect data will then have to be connected to the database [3]. The connection will allow the database to have constant read and write operations from the application. The application is where data will be retrieved and collected. The only other manual operation that the user has to control is the data back up. Users will have to manually backup data and implement other features through AWS applications. Most other processes will be handled on the back-end and are provided by *AWS* such as security, access control, data management, monitoring and many other features.

Many implementations of this database will have to be modeled and implemented based on a specific circumstance. If a user is planning on running an application that has many concurrent users and big amounts of data then the user will have to make a database fitting the criteria for their needs. The same could be said about applications with other types of characteristics. We believe modeling and implementing databases must be individualized to the needs of the data that an individual or company is handling. This can also be said for any database development, whether it being No-SQL, SQL or other approaches. Regardless, a local developmental version of DynamoDB is now available that does not require development on the Web API but does require a connection to *AWS*. DynamoDB uses JSON for its syntax because of its ubiquity. Although, most programming languages and frameworks work with a DynamoDB binding.

2.5 Queries

There is not a very robust querying range within DynamoDB. With the use of a primary key, the user can query for many different items or tables within the database. Composite primary keys that consist of a partition key and a sort can are very useful when querying. Most of the time, user will use a sort key to gain an extra layer of querying. That is about all the sufficient querying options within DynamoDB. Unfortunately, this is also where DynamoDB lacks in comparison to some other No-SQL databases. Many have tried to query with the use of attributes to gain an extra layer of specification, but that just proved to be detrimental to the performance of the database. Failures occur or retrieval will take an unexpected amount of time. Overall, querying is not one of DynamoDB's strongest factors.

2.6 Best Use Cases

DynamoDB does offers some great benefits in specific cases. If a user needs a database to just throw some data in for a smaller, one-off or personal, project, then this is the database for that user to implement. DynamoDB works very well with prototyping data in large amounts or small amounts so if the user needs a database to research some ideas then DynamoDB is a great choice. Applications that require predictable, low and constant amount of data handled will benefit well from this database. DynamoDB does not work well with unbounded and fast data growth so user tend to stray away from it in those cases. Users that need to have deep queries, stray away from DynamoDB since the query range is very small.

3 Couchbase

Couchbase is an open-source, distributed, document-oriented No-SQL database. Is is designed to provide scalable key-value or JSON document access. It was designed to offer the speed and scalability of the designers' previous product, Memcached, but with the robust querying capability and data persistence offered by a database management system. The open-source nature of Couchbase's code allows for easy developer integration as libraries are provided for several programming languages including Java, .NET, C, Python, and others. The data model itself is flexible as JSON documents can act as a nested structure storing more documents as well as binary attachments. The data model is designed to be automatically scalable with little input needed from the database manager. Couchbase uses sharding and clustering to scale a database to be able to handle a growing and/or shrinking level of throughput while also maintaining a high level of performance when tracking real-time user activity.

Within the next subsections, we will discuss how data is modeled and managed in Couchbase, as well as Couchbase's performance, implementation, querying, and finally we'll cover the best use cases of Couchbase.

3.1 Data Modeling

As mentioned in the section introduction, Couchbase is a document-oriented No-SQL database. As such, the main unit of data is known as a *Document*. As with many database structures, data in Couchbase are stored as a key-value pair, where the document key (or *Id*) is an immutable string of up to 250 bytes and the value is either a JSON object (a *Document*) or a binary object (*blob* or *attachment*).

Each *Document* in a Couchbase database must have the following required attributes: a unique document ID, a current revision ID (which changes each time the database is updated), the history of all past revision IDs, a body (in the form of a JSON object). The document ID must be unique, as it is the primary identifier of the *Document*. The current revision ID stores a value that is unique to the *Document*, meaning it cannot already exist in the revision ID history. This allows the database manager to track any changes that have happened within the database using version control similar to Git. Any conflicts in the database can be tracked and resolved, synchronization is more smooth, and reverting to a previous version is supported directly. Finally, the body of the *Document* is itself a set of key/value pairs, and can use this characteristic to act as a nested data structure: JSONs within JSONs.

In addition to the four required attributes above, a document *may* have one or more named binary *attachments* (or *blobs*). JSON objects can store particular textual data types: integers, strings, booleans, and JSON nulls. As these types can be stored directly in the body of the *Document*, there is no need to create an *attachment* for them. *Attachments* can be used to store any non-textual data such as media files. These files are stored as *binary large object* (commonly abbreviated to *blob*), which exists inside the database structure as a persistent value. *Blobs* are commonly used to store images, audio, video, or other multimedia files, but in reality can be any collection of data stored in a binary format.

3.2 Data Management

One of Couchbase's most convenient features is the automatic scalability that is offered through the use of server clustering and data sharding. These two powerful methods of horizontal scaling allow for the number of servers in use to increase or decrease depending on the current data throughput requirements as well as the number of active users manipulating data simultaneously.

As Couchbase is a distributed database, data can be stored on a *cluster* of multiple servers connected by a network. The advantages of distributed databases and clustering include horizontal scaling as previously mentioned, high availability, and the replication of data in secondary servers to prevent full data loss. Horizontal scaling can be simply defined as the addition of more resources (in this case more servers), as opposed to vertical scaling, which would be the replacement of whatever current servers your database is hosted on with more powerful ones. One reason Couchbase can boast high availability (i.e. minimal downtime) is due to the distributive nature of clustering. For example, if some data is currently stored on Server A only, then any time Server A goes offline, the database is unusable. However, if data is stored on *primary* Server A and replicated to *secondary* Servers B and C, then if Server A goes offline, Server B can be designated as the new primary server and Server C can begin receiving the replicated data from Server B instead. As long as at least one server with up to date data is available to be promoted to the primary server, the database will never be fully offline. This also provides a safeguard against full data loss, as there should always be at least one server online with accurate data.

While clustering is useful for providing high-availability, *sharding* is useful for handling increased load. This technique splits data into "shards" that are stored on different servers (or nodes) within a cluster. Additionally, each node stores replicas of a few other shards as backups as additional fail safes. A simple example: Node 1 stores all of Shard A's data and half of the data on each of Shard B and C, Node 2 holds all of Shard B and half of A and C, and Node 3 holds all of Shard C and half of A and B. If Node 3 goes offline, the data is safe, as Shard A is stored by Node 1, Shard B is stored by Node 2, and the entirety of Shard C is split and stored on Nodes 1 and 2.

A key point that makes these already useful techniques even more user-friendly is that Couchbase is able to create server clusters and shard the data automatically. Due to this, if there are issues with server loads, more nodes can be automatically added and the data will be redistributed without any manual input required from the database manager [4].

3.3 Performance

The performance of Couchbase databases depends on many factors, including the size and type of data stored, the number of concurrent users accessing the data, what types of access are most commonly used (e.g. reads, writes, updates), among countless others. The high level of scalability offered by sharding and clustering was covered in the previous subsection, but is also relevant here. In general, the performance of any data structure is highly influenced by the size and transfer rate of the data. However, Couchbase and other distributed databases strive to minimize the impact of these variables by providing options for automatic horizontal hardware scaling [5]. While no database structure is *infinitely* scalable, having the ability to maintain high performance under heavy loads is a benefit that Couchbase does offer.

There are some things that Couchbase could do better, however. For example, the overhead

per document is larger than some other document-based No-SQL options, which can become a significant issue when dealing with a large number of small documents. If the database is primarily working with larger documents, the overhead will be negligible by comparison and should not noticeably affect performance. However, working primarily with larger documents can create its own challenges, such as the use of more resources to shard this data and replicate it to multiple servers within clusters. Another potential downside is the learning curve associated with the proprietary querying language, *N1QL* (pronounced "nickel"), which will be covered in a later subsection, Querying.

3.4 Implementation

Couchbase's open-source nature means that its implementation depends significantly on the intended use. The database can be structured based on need, in part due to the flexibility of the N1QL querying language. The developers of Couchbase have kept up to date with advancements in common SDKs, and as such there is continued support for many languages, including Java, .NET, GO, C, Node.js, Python, Scala, Ruby, and PHP. Once the database manager has added the data to the database via their preferred method, the scaling and sharding of the database can be fully automated by Couchbase. Live analytics determine the number and size of clusters to use based on number of operations in a given time frame.

An advantage of Couchbase being built around the JSON data structure is that JSON data can be easily retrieved and manipulated via client-side operations in commonly used languages like jQuery. Having easily accessible metadata is beneficial for analyzing the efficiency of a database, as well as providing a solid jumping off point for post-processing, data visualization, and generally getting the most out of the data being stored.

3.5 Queries

As mentioned in the Performance subsection above, querying in Couchbase uses a proprietary query language called *N1QL*, rather than a more commonly used language. This approach can lead to challenges when a database manager is attempting to build a database in Couchbase for the first time. *N1QL* uses a non-first normal form (N1NF) data model, from which it derives its name. Couchbase announced *N1QL* in 2015, marketing the language as "SQL for documents", and notes that *N1QL* is a super-set and generalization of the first normal form, which we know is used in the relational model [6].

This data model is a dialect of SQL with more features focused on querying and manipulating JSON data. Its syntactical similarity to SQL when querying can make for an easy transition from more traditional database models, however the JSON structure in which the data is stored may be unfamiliar to those who are used to seeing the SQL structure. Some advantages of *N1QL* include built-in array functions such as *MAX()*, as well as additional keywords and functionality beyond those offered by SQL, allowing for better access to data that lives deep within nested data structures. We've discussed the ability for Couchbase to store JSONs within JSONs for conveniently nested data, so it does make sense for the developers to create a language that allows for even more efficient querying of those structures.

3.6 Best Use Cases

The burning question on most database managers' minds when deciding which database option to choose is: "which database is best for my particular data-set and user base?" Three key situations where Couchbase is likely to be a good option are when dealing with document-heavy data, when dealing with many concurrent users performing multiple operations, and when horizontal scaling is required. Of course, if only one or two of these are true, there may be better options for a particular need. However, if all three situations above are relevant to a need, then Couchbase is likely among the best options available.

Document-heavy data is what Couchbase was designed to handle. There are many specialized tools that are relevant to storing a large number of documents, including text caching, full-text searches, and document analytics. It should be unsurprising that the creator of "SQL for documents" is responsible for creating a powerful document-based database. The second situation above refers to when many users are performing concurrent full-text searches or N1QL queries, causing heavy server load. Couchbase can automatically scale up the number of servers to handle the increased load, and then scale the number back down when traffic has reduced in order to conserve resources. Finally, applications that require horizontal scaling often have a common concern: having a single point of failure. As explained, Couchbase can alleviate this worry due to the automatic sharding of data. This ensures that if one server or server cluster fails, the data is distributed among the remaining servers so there should never be complete data loss under this structure. In general, Couchbase has a wide range of use cases, especially due to the code being open-source, however it is far from a one-size-fits-all solution.

4 REDIS

REDIS is an open source, in-memory, and key-value No-SQL data structure store used as a database, cache, message broker, and streaming engine. The name is derived from "Remote Dictionary Server" and was developed and distributed by *REDIS* (formally known as *REDIS Labs*). REDIS offer high scalability, enhanced geospatial data and in-memory data structure for performance enhancement. REDIS is famously known to be used as a cache system. Originally created to improve scalability in a small Italian startup. REDIS is the most popular in-memory data structure server.

Within the next subsections, we will cover how data is modeled within REDIS, how data is managed, the performance of REDIS, how queries are implemented and their robustness, how users can implement this database and the best use cases for this approach.

4.1 Data Modeling

Let's first discuss how data is modeled with REDIS. This database uses a key-value data structure that is much similar to DynamoDB where all data is denormalized. This is where the similarities to DynamoDB stops because the data structure used in REDIS are much different. The use of data structures allows for data modeling to be fairly straight forward as long as the developer gains a great deal of knowledge on each data structures and learns when and how to use them.

Developers will use multiple, sophisticated data structures that allow for storage, organization, processing and retrieval of data. These data structures come in the following formats: Strings, Hashes, Lists, Sets, Sorted Sets, Bitmaps, Bitfields, HyperLogLog, Geospatial indexes, and Streams. Many of these data structures are very complex and will have a learning curve compared to any other database approach [7]. All data structures are useful in different situations and all have various various use cases. For example, if you want to create a system cache, then you would most likely use a String data structure. Strings, lists and hashes should be fairly similar to the traditional computer science interpretation of each term. Only one data structure can be implemented per session. Each data structure uses a key to uniquely identify it. This key will be important to the management of the data and querying. Data structures also have various commands that can be used to delete, type, rename, query and much more.

4.2 Data Management

Since REDIS is used for most caching situations, lets first talk about how caches operate. Caching systems generally store data in fast access hardware such as random-access memory (RAM). This is the reason why REDIS is classified as a in-memory database. The cache can be thought of as a high-speed data storage layer which only is stores small subsets of data rather than a full data set. Many times caches are used alongside another primary database to increase the data retrieval performance and store low-latency data. A properly implemented cache system will have a high hit rate with little misses present for proper data retrieval. If a miss occurs, then the application will retrieve the data directly from the main data store. Memory caching stores the "hot" data within a database.

REDIS can also implement multiple clusters of data that allow for data replication. There are two main options that can contribute to REDIS. The two options that allow for availability and scalability are called primary-replica architecture or a clustered topology. Primary-replica architecture refers to a type if master-slave architecture but with primary nodes and replication nodes. Clustered topology is defined by the type and state of each node in the cluster and the relation between them. Both of these are offered through a service called *REDIS Cluster* which is also open source [8]. REDIS cluster uses multiple master and slave nodes to distribute to create a clustering of nodes effect similar to DynamoDB. Data will be automatically sharded across the multiple REDIS nodes that are created in the cluster. This process is very similar to Couchbase. We will not dive into exactly how partitioning, replication and sharding occurs within the cluster because we covered how they occur in the DynamoDB and REDIS sections. The same process for all dynamics occurs within REDIS cluster.

Data management in REDIS is mostly handled by the user. With a proper implementation of data structures, the user will have little to no problems managing data regardless of the approach of REDIS. It is very important that the user understands how to correctly implement the data structures to manage their data.

4.3 Performance

REDIS has the reputation as the most popular caching system used as of right now and it is for good reason. This approach handles data very well and its performance is immaculate. Most times, there is little to no time loss when using REDIS as a cache. In-memory data stores are known to be very fast with average read and write operations. In fact, most read and write operations take less than a millisecond to compute even with very high traffic and concurrent operations being processed [9]. Other, non-cache implementations are known to perform well also. Since REDIS is a in-memory data store, there are many performance advantages to using this implementation.

Some downsides to REDIS performance include a slowdown in operation speed when sharding occurs. Since REDIS has a master-slave approach to deal with sharding, a disaster can occur when a master node is down. Many people have reported data loss due to a master node being down. Since we are on the topic of node clustering, there can be overhead when users do not properly respect the cluster topology and the states of all the nodes. Lastly, there requires a extremely large amount of RAM when implementing REDIS with large applications.

4.4 Implementation

REDIS works on most POSIX systems like Linux, BSD, and MacOS X, without external dependencies. Most people and REDIS themselves recommend using Linux for deployment. The user will write scripts to handle data through data structures, contribute to the write and read operations and REDIS will handle all back-end consistency and data management operations. You can use REDIS from most programming languages but is normally written in ANSI C. Implementation of REDIS will be directly correlated with the use case, just like DynamoDB and Couchbase. The user will have to implement a very sufficient data model and without one, there can be catastrophic issues with the database. REDIS is fairly straightforward and easy to implement.

4.5 Queries

Queries are not very robust within the REDIS environment. There is a select amount of filters for querying such as tag filter, numeric filters, geographic filters and much more. Querying syntax is also a difficulty when it comes to querying within REDIS. The syntax is simple, but very different from any other querying language that we are used to. The REDIS website has told us that there are many way to compute complex queries, but through many other sources, we found this not to be true. In fact most people know that there is no good way to query deeper. Since REDIS is a key-value store, we can make a conclusion that many of the querying capabilities will be lacking, which was a problem with with DynamoDB as well. Overall, querying is not implemented well and there is no way to implement high leveling querying within REDIS.

4.6 Best Use Cases

REDIS is best used in a caching scenario. In fact, we have seen little to no implementations that were not used a cache system. If a user would like to improve their database performance then this system is the best option for them. Data structures also have very robust geospatial data storage options so if a user wants to implement geographical and spatial data then this is a great option. Many machine learning applications, chat and messaging, and targeting and personalization sites have also used REDIS.

5 Conclusions

DynamoDB, Couchbase and REDIS are all very useful No-SQL database approaches. All three of the databases have very different uses of data modeling, management of data and many other features which helps each approach useful for certain tasks. There are many advantages and disadvantages to each database implementation. We can expand this idea and say that there are many ways to implement No-SQL databases besides the three we spoke about in this paper. Each No-SQL database can be useful, but in specific circumstances. Some databases are better for caching, some offer high consistency and others may value availability in much more scenarios. Even though many of these databases are different, they all do have the same ideals about characteristics of the database. For example, scaling is very important to how different systems operate and perform and is one of the most spoken about factors in No-SQL databases. We believe that one of the most important part of implementation of a system is modeling data to your specific use case. Poor data modeling could lead to a loss in time and money.

All of the systems that we spoke about in this paper could not be properly compared due to the various different advantages and disadvantages of each system has in certain scenarios. DynamoDB has the advantage when it comes to implementing a consistent database for small and large amounts of data. Couchbase has a very robust document-based data structure which applications that mainly deals with documents could benefit much from. Lastly, REDIS is one of the most used and high performing caching systems that is offered. Overall, we learned that there are many No-SQL data stores that all have their advantages and disadvantages but are extremely useful in certain cases. Anyone that is attempting to deploy a No-SQL database should do their research into which system suits their application the best.

References

[1] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels, "*Dynamo: Amazon's Highly Available Key-value Store*", ACM SIGOPS Operating Systems Review, Volume 41, Issue 6, 2007, Pages 205-220.

[2] Tanmay Deshpande, "*DynamoDB Cookbook*", Packt Publishing, 2015, Pages 1-29, ISBN 978-1-78439-375-5.

[3] Niranjanamurthy M, Nitesh S N, Nagesh S N, Balaji Sriraman. "*The Research Study on DynamoDB: NoSQL Database Service*", International Journal of Computer Science and Movile Computing, Volumn 3, Issue 10, 2014, ISN 2320-088X

[4] Tiwari, Shashank. "*Professional NoSQL*", Wiley Pubishing, Inc., Indianapolis, IN, 2011, Pages 274–274.

[5] "Query: Fundamentals." Query: Fundamentals — Couchbase Docs, https://docs.couchbase.com/server/current/n1ql/query.html.

[6] "N1QL Language Reference." N1QL Language Reference — Couchbase Docs, https://docs.couchbase.com/server/current/n1ql/n1ql-language-reference/index.html.

[7] Shanshan Chen, Xiaoxin Tang, Hongwei Wang, Han Zhao and Minyi Guo, "*Towards Scalable and Reliable In-Memory Storage System: A Case Study with Redis*", 2016 IEEE Trustcom/BigDataSE/ISPA, 2016

[8] Tiago Macedo and Fred Oliveira, "*Redis Cookbook*", O'Reilly Media Inc, 2014, Pages 10-24, ISBN 978-1-449-30504-8.

[9] Jeremy Nelson, "*Mastering Redis*", Packt Publishing, 2016, Pages 153-191, ISBN 978-1-78398-818-1