SQL or NoSQL: A proposed database solution for the BeaverCoffee system

Betty Brändström
Arlöv, Sweden
BettyBrandstrom92@hotmail.se

Casper Strand
Malmö, Sweden
casperstrand@outlook.com

Lars Korduner
Malmö, Sweden
Korduner.Lars@gmail.com

Lily Hammerskog
Malmö, Sweden
Lily.Hammerskog@gmail.com

Mattias Sundquist
Arlöv, Sweden
mattias.sundquist@gmail.com

***Abstract*—This paper investigates the four major implementations of non-relational databases, in an effort to assess the most appropriate one for the use cases required by BeaverCoffee AB, as well as comparing it to what a traditional relational model would look like.**

Each data store is described, analyzed and provided with a suggested data model in the form of a UML diagram. These are then compared in full to propose the most appropriate data store for BeaverCoffee's use cases, which ended up being MongoDB, with a detailed discussion as to why MongoDB was found the most appropriate data store BeaverCoffee's system.

The data store is then used for the implementation which is included alongside this document. Installation instructions can be found in the Appendix.

I. Introduction

When it comes to processing and storing large and complex data sets, traditional RDBMS (Relational Database Management Systems) are usually not an optimal solution. For modern applications generating big data sets with a diverse structure, a non-relational database system would likely prove to be the more ideal implementation [1].

A. Non-relational concept

As the name implies, a non-relational database does not follow the relational model that RDBMS are designed around, which increases their scalability and flexibility. Unlike their relational counterparts, most of them are designed to handle large-scale amounts of data by running on a widespread cluster of smaller and cheaper commodity machines. This makes them appropriate for workloads that handle large amounts of data in applications such as IoT (Internet of Things) devices, E-commerce and Social Networks [2]. Additionally, the non-relational databases does not require a pre-defined schema or the traditional tabular schema of columns and rows. Instead these are replaced by other models that are designed to handle various types of semi-structured and unstructured data.

B. NoSQL platform

More commonly, non-relational databases are referred to as NoSQL (Not Only SQL) databases, of which there are four main database types. Each of them are designed to efficiently process and store data in their own way. A brief introduction for each database type, their primary functions and use cases, is presented below.

- Key-Value stores. When the access to data within a database is made through the usage of primary keys, key-value stores are most likely to be the best implementation. The data is structured as a simple hash table of key-value pairs where the keys are unique, either user-defined or generated, and the value may be data of any type. Common use cases for key-value stores are storing session information, user profile, preferences and shopping cart data [3].
- Document stores. Just like the name suggests, a document store is the go-to implementation when large readable data sets need to be processed and stored. It can be considered a specialized key-value store where the value is an examinable document, usually JSON, XML or BSON-file. The documents that are stored are usually similar, but do not have to be exactly the same and may contain different attributes. The most common use cases are content management systems, blogging platforms, catalogs and event logging [4].
- Column-Family stores. Unlike the traditional RDBMS which store data in rows, the data in column-family databases are stored in groups of related data called column-families. The data contained inside the family is usually accessed together. The column-family consists of multiple rows, each having its own set of columns associated with the row key. The number and value of columns may also differ between the rows, as the columns do not span all the rows as they do in a RDBMS [5]. Typical use cases for this type of database would be event logging, content management systems, blogging platforms and counters [6].
- Graph stores. When the connections or relationships between data is equally important as the data itself, a graph store would prove highly efficient. In graph databases data is stored as entities (nodes) and relationships (edges) between those entities, which both may have multiple properties [7]. The edges have directional significance and organizes the nodes into patterns defined by their

relations. All defined relationships are persisted rather than being calculated at query time, which allows for quick traversals. Use cases where graph stores are particularly useful are connected data (e.g. social networks), recommendation engines, routing, dispatch and location-based services [8].

II. Description

A. Background

BeaverCoffee is a company that provides customers with high quality brewed coffee at physical locations, as well as coffee beans. The demand for their product is high, and there are expansion plans in place to expand business to many locations worldwide. In order to support this expansion, BeaverCoffee requires a database structure that allows for easy expansion to different sites while keeping things consistent across the company.

B. System context

The system that will be developed is intended for internal use only. Customers only act on the system indirectly through an employee or separate online infrastructure. There will be several types of agents that interact with the system. Regular branch employees are the lowest level, and are limited in their scope. Managers of different levels have higher privileges and functionality. While not in scope for this project, the system is developed to allow for a web server to act similarly to a user, allowing for future development of a web-based store. The system will be able to handle customer orders, customer information, employee information, stock counts, product information, reports, and detailed records for accounting purposes.

The sequence diagram in Fig. 1, illustrates the basic functionality, interactions and boundaries between entities in the system. As explained above, an operator is an employee who may interact with the system in a manner appropriate to their employment status. The system in turn queries the database for read and/or write operations based on the operator input. The following subsections presents two different implementation models for a viable database solution, as well as discusses their strengths and weaknesses. The first is designed around the traditional relational model, and the second is designed to support a NoSQL implementation.

C. Relational model

In a relational database, the first step is typically to construct a schema defining tables with columns and rows, along with relationships between them. In Fig. 3 a basic schema for a relational model is presented as a ER (Entity Relationship) diagram using crows foot notation [9]. For a company like BeaverCoffee that is still growing, a model that can handle changing application requirements is necessary. With a predefined schema and structure, the relational model is limited in its flexibility compared to its non-relational counterpart. If the relational database is to update its schema design, the process can be very slow and cause application downtime as the data is being migrated into the new and updated schema.

Another limitation of the relational model is its ability to scale. As BeaverCoffee is expanding and gaining more popularity, it will be needing a system that will be able to handle a heavy workload. Relational models scale vertically,

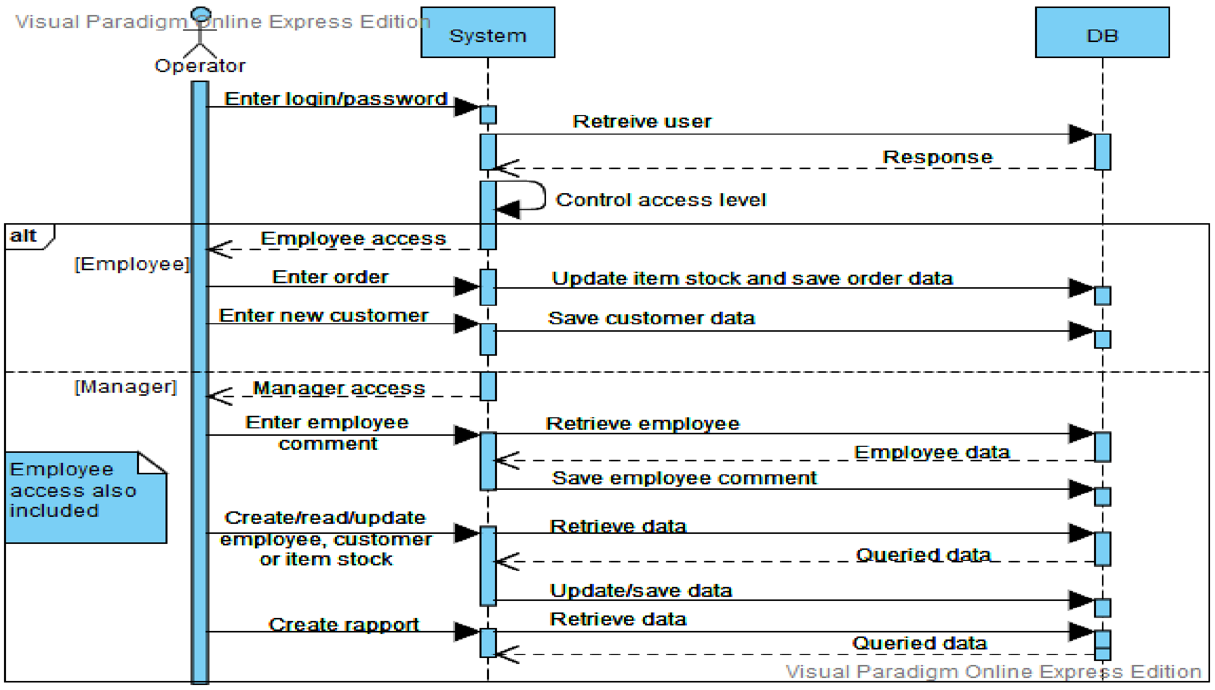

Fig. 1. High level UML sequence diagram.

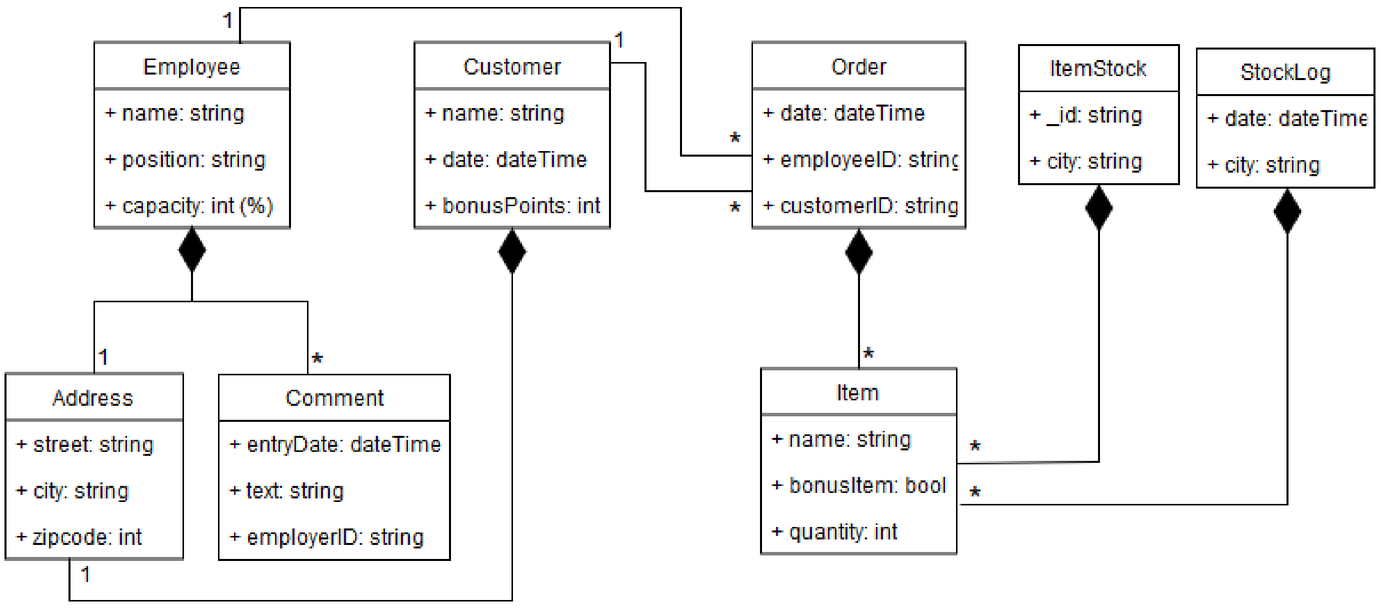

Fig. 2. UML diagram demonstrating NoSQL aggregate model

which means that they run on a single server, hosting the whole database. In order to properly scale, investments in better and expensive servers would have to be made, making the horizontal scaling of NoSQL models a more preferable solution for the BeaverCoffee company [10].

D. NoSQL model

One of the main advantages of NoSQL databases is that they allow flexible and dynamic schemas. As mentioned in the description background, BeaverCoffee is a company that is rapidly growing and expanding, which makes this feature highly relevant. The flexible model will enable the company to easily make changes to schemas while also avoiding application downtime, a feature not supported by relational data models. Another key advantage of the NoSQL model, relevant for the BeaverCoffee system, is scaling. Instead of scaling vertically as relational models do, it is done horizontally. Generally resulting in better performance as the data workload is distributed across multiple servers [11].

In the diagram in Fig. 2, the database model is demonstrated using UML notation (to limit the size of the diagram, not all data fields have been included). Like most NoSQL implementations, it follows an aggregation-oriented model which is illustrated using the black diamond symbol and 1-* to describe relationships between data. The aggregation model contributes to high performance reads, usually outperforming the relational model, which typically needs to perform joins on different tables to retrieve the equivalent data. It also makes it possible for the related data to be updated with a single atomic write operation, keeping it consistent throughout the database [12].

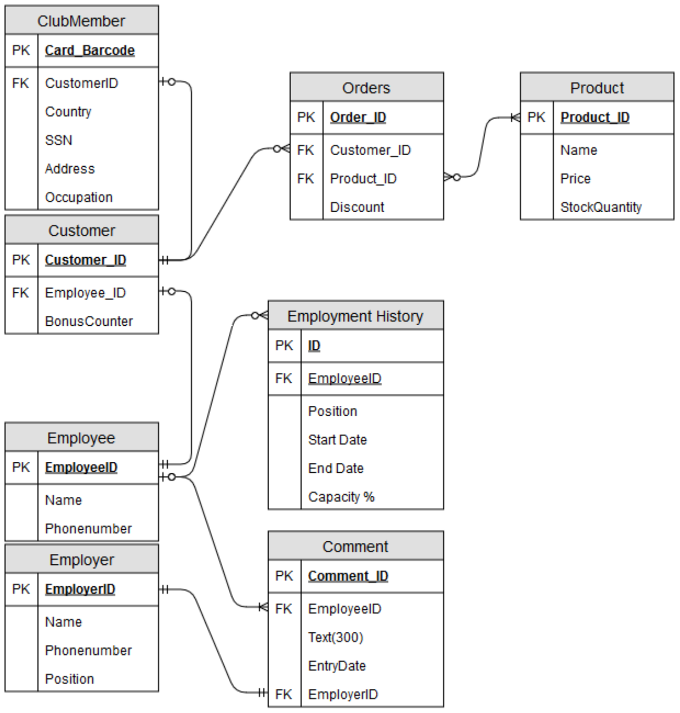

Fig. 3. ER diagram illustrating a relational schema

III. Analysis

A. Neo4j - Graph-store database

Graph databases has been used widely in social networking sites. This is because in graph database modeling, an emphasis is put on the relations between different entities. It focuses on the relationships since it wants to be able to handle so called three or more depths relations, such as a friend of a friend

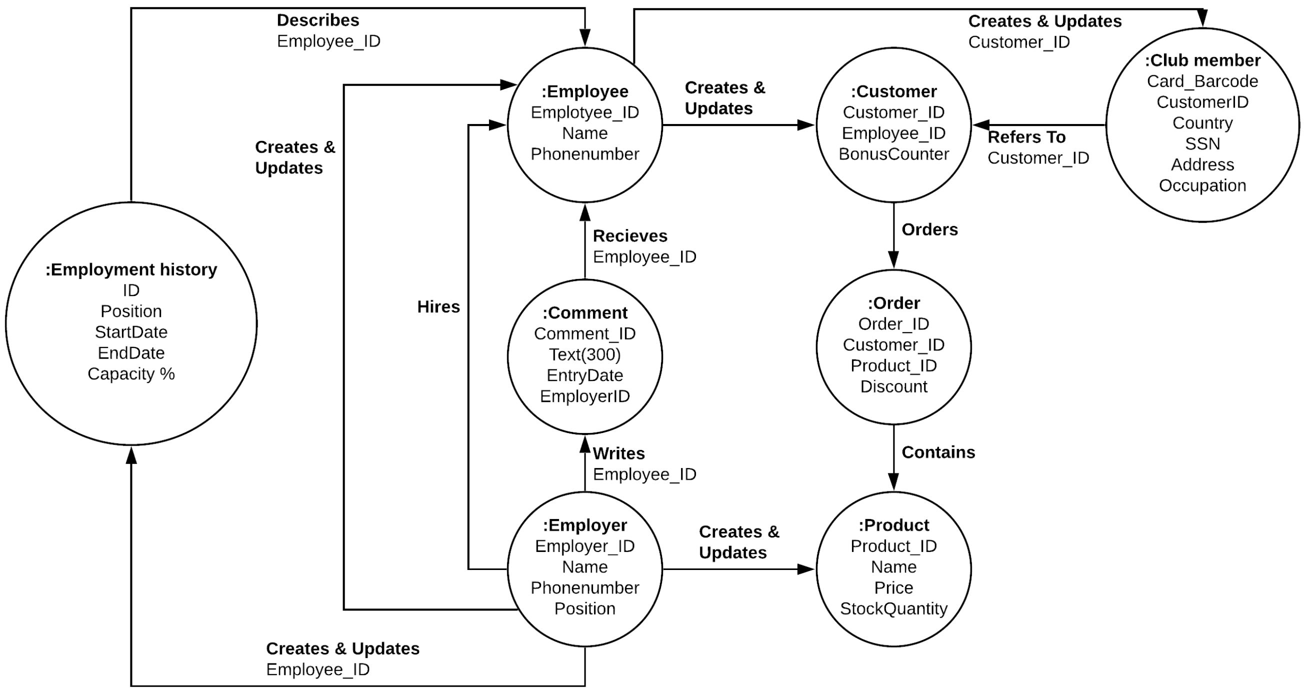

Fig. 4. Graph-based database diagram

of a friend on a social networking site. A relational database can not handle this in an acceptable time frame and therefore looses out to the graph database model in these situations [12].

When handling big data one needs to be able to update parts of the database without a need to lock down the rest. This is easily done in real time in graph databases such as Neo4J. In Neo4J you can update the database by adding or removing new relations and entities from the database whilst running queries since a graph store only touches the relevant data as it is running its queries. In a relation database there is the assumption that any query will touch the majority of the data which locks down the database when designing new relations between the data. Although this is not really needed in the BeaverCoffee system since it is fairly complete in how it is designed and does not need to be able to add or remove many more entities at its current state [12].

In the BeaverCoffee system relational queries could be used to more easily handle business intelligence issues, such as retrieving a list of customers based on what other customers has ordered together with a specific product. This could then be used in advertisement at a later stage [12]. But since this is not something that is needed at the current marketing phase of the BeaverCoffee franchise it was concluded that a graph store would not be used.

One negative point when working with graph-based databases is that there is currently no standardization on which the query language is based on [12]. This makes it harder for programmers to widely use the graph-based database model for querying since they would need to learn a specific query system for the BeaverCoffee database, which is not a good idea since the company is still early in its development.

A proposed graph-based database model diagram for the BeaverCoffee system can be found in figure 4.

B. MongoDB - Document-store database

Main qualities of a document-store is its ability to scale and adapt. Document databases are flexible and semi-structured, enabling them to change and evolve with the shifting needs of a system application. The main requirements of the Beaver-Coffee system is that the solution is supported by a database structure able to handle a growing business expanding to locations worldwide. Being both flexible, scalable and widely used for use-cases such as catalogs, user profiles, and content management systems, a document database would prove a top candidate that would ease the expansion of the BeaverCoffee company. For the implementation of the database, the leading name in document-stores was selected.

MongoDB is a NoSQL, cross-platform, document-oriented database program. While SQL database systems uses tables of rows and columns to structure data, document databases such as MongoDB uses documents. In MongoDB, documents are stored in a collection, which in turn makes a database. Documents are analogous to rows in a SQL table, but there is one big difference: not every document needs to have the same structure, each of them can have different fields, which can be a very handy feature in many situations. Another feature of MongoDB is that fields in a document can contain arrays and

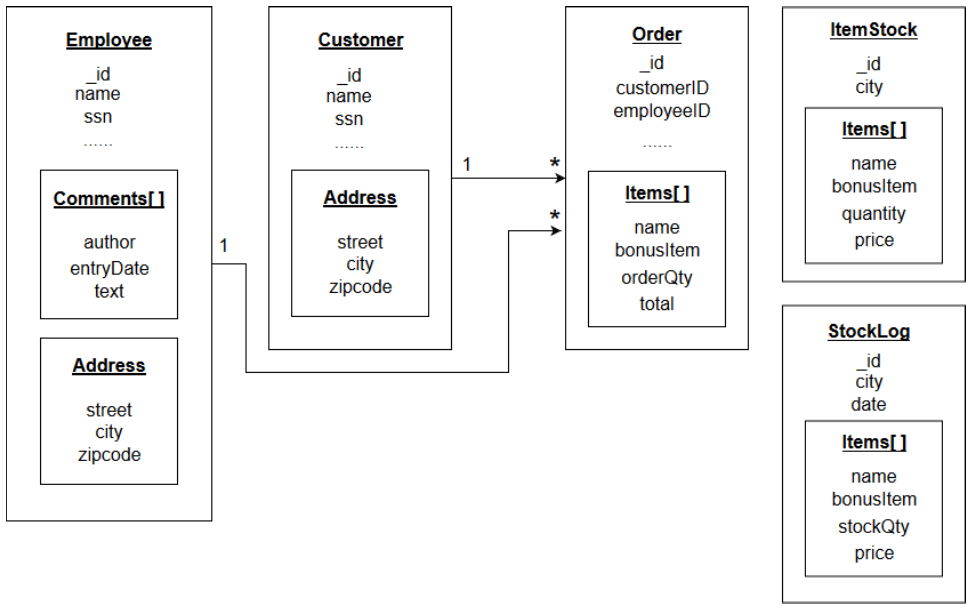

Fig. 5. Diagram illustrating a document database model

or sub-documents (often referred to as nested or embedded documents). Each document in a collection in MongoDB can look completely different and how the structure of a document is designed is up to the developer. MongoDB does not enforce a schema, but the system application should. Although MongoDB is generally high performance, the structure and index of the documents and collections have a big influence on the performance of the application. While designing a schema one should focus more on how the data is inserted, updated and queried and less on how the data is structured.

MongoDB stores documents with the use of a BSON protocol. BSON is a protocol for binary serialization of JSON-like data. JSON is a data exchange format that is popular in modern web services. It provides a flexible way to represent complex data structures. BSON is compact and in most cases storing a BSON structure requires less space than its JSON equivalent. BSON provides additional data types not found in regular JSON, such as Date and BinData. One of the main benefits of using BSON is that it is easy to traverse. BSON documents contain additional metadata that allow for easy manipulation of the fields of a document, without having to read the entire document itself. The BSON implementation enables MongoDB to become lightweight, fast and highly traversable. MongoDB provides software drivers for most modern programming languages. The drivers are built on top of the BSON library, which means that interaction with the MongoDB database are often done directly with the BSON API when building queries. MongoDB supports five categories of queries: Connect, Insert, Find, Update and Delete, where each major category often has multiple subcategories i.e: InsertMany, DeleteMany etc.

In Fig. 5, a simple diagram demonstrates the planned structure of the document database. It is designed to consist of five main collections; Employee, Customer, Order, ItemStock and StockLog. The diagram also shows embedded documents with related data nested inside the main documents. Because of the fact that a document can be placed inside another, one is able to access the embedded data in a single query to the database. This results in fast reads, usually outperforming the relational model, which typically needs to perform joins on different tables to retrieve the equivalent data. It also makes it possible for the related data to be updated with a single, atomic write operation, keeping the documents consistent throughout the database.

Also illustrated in Fig. 5, is document referencing. For example, the ID of employee and customer is used as a reference inside the order document. Implementing referencing has the downside of having to perform secondary queries application-side in order to access the referenced data. But this kind of manual referencing is necessary to properly maintain the relation between data placed in separate documents [13]. The decision of when to use embedded or reference design is primarily based on the queries (inserts, updates, etc.) that is to be made to the database. Current data model, a combination of both designs, is used to access data in a way most suitable to the functionality of the BeaverCoffee application.

As CAP (Consistency, Availability, Partition-tolerance) theorem dictates, a database can only fulfill two standards at the same time. MongoDB is by default strongly consistent and partition-tolerant. The consistency is achieved by the use of replica sets, each containing a primary node (master) with a set of secondaries (slaves). Writes are made to the master which then is replicated to all or a number of slaves before returning successful, making sure that all slaves are properly updated and reflecting the data in the master. Partition tolerance means that the system can remain operable during intermittent network failures, which is also achieved through the use of replication and replica-sets. Data is always sufficiently replicated across multiple servers, enabling the system to function despite network partitioning. However, the assumption that MongoDB entirely sacrifices the second CAP criteria, availability, is incorrect. When a master for some reason fails and goes offline, MongoDB actually provides high availability through automatic failover. The automatic failover process basically means that if slave nodes in a replica set has lost communication with the master for a specified amount of time, a new master is selected amongst the slaves based on their priority configuration. When the old master comes back online, it quickly catches up and joins in with the other slaves. While the election is being made, write operations to the replica set cannot be processed, hence the sacrifice of complete availability. Despite this, MongoDB still qualifies as a highly available document store, ensuring high consistency and availability for the BeaverCoffee system.

The scaling of MongoDB is done horizontally by the means of a technique called sharding, where the dataset is partitioned and distributed across multiple different servers. Each shard can consist of a replica set, keeping the shard available on multiple nodes. When data is partitioned, it is important that the data is clumped together in a way that allows the user to access wanted data by visiting a single server. I.e. an obvious task for aggregates. If implemented properly, sharding significantly increases the read and write performance of an application. Also beneficial for performance and relevant to the BeaverCoffee system, is how data is arranged on the nodes. As the company is expanding to locations in different countries, the data can be distributed based on geographical zones, kept as close as possible to the user. For example, with one datacenter in America and one in Europe, serving the clients closest to their location.

In summary, a document database such as MongoDB presents a number of favorable attributes when it comes to supporting an application with requirements like BeaverCoffee, and is therefore the data store recommended for the system.

C. Cassandra - Column-store database

Columnar databases are primarily useful for large scale data analysis. They excel at storing large amounts of data with high compression, and at reading large amounts of data quickly. Relational databases are not optimized for the analytical use case, as fetching large amounts of data of a single type is a costly process. An example would be to examine the latest time all users were logged in to get data such as peak active users and user retention. When dealing with hundreds of millions of entries, a relational database would struggle to get that data within an acceptable time frame.

The issue with columnar databases for this specific use case is the heavy emphasis on read and storage efficiency. The most glaring weakness of a columnar database is the writing performance. It is a costly process to enter a new entry into a columnar database, due to its structure. A row based structure can enter a new record with a single operation, while a columnar database must enter into each column individually. In order to preserve transactional integrity, all the write operations for each column must be entered before trying to enter a new record. The fact that each column must be queried individually to retrieve data related to a record means that efficiency is low when referencing multiple columns at once.

Modern solutions such as Cassandra ameliorate some of these issues, but not all. There are still large inefficiencies with write operations, especially when dealing with transactions. Updating multiple columns requires multiple operations, which causes issues when dealing with multiple column families. Cassandra is exceptional for analytical use cases, but is sub-optimal for transactional ones. In the case of BeaverCoffee, we need the database to be able to easily and quickly create and update records across multiple tables, or column families in the case of columnal databases. Using batched processing can help ameliorate the writing issues somewhat, but causes unacceptable delays between a customer interaction and a database update.

Cassandra, or any other columnal database, might have certain use cases for the organization. One example would be analytics for sales metrics and advertising purposes. This would need to be a separate database however, as the transactional issues with day to day operations are prohibitive. A possible structure for this would be to implement a separate columnal database that gets data from the primary transactional database in infrequent batches, which is then used to aggregate data for monthly or quarterly reports. This solution might not be necessary in the short term as other database structures will perform adequately for analytics when dealing with a medium scale operation.

D. RiakKV - Key-value store database

Key-value databases, like RiakKV, emphasize high availability at the expense of strong consistency. In a realistic scenario this means that the database stays available for writes and reads, regardless of whether it has severe network or machine issues, however this comes at the expense of not knowing how accurate or up to date the data is when a couple of machines are down [14]. Which is useful for a number of scenarios where you want to keep collecting information [15] [16], like a users internet browsing session through cookies, and take care of any discrepancies with the data later [17]. This is because the database is in reality distributed to multiple nodes in a cluster, which means it has a ”eventually consistent” data model. Requiring the nodes to sync up to a set level of consistency, where a set number of nodes have the same data, before being considered consistent [14].

There are several nodes that all should contain the same data in a perfect scenario, with the software speaking through a loadbalancer that assigns retrieval and writing tasks in an appropriate manner to the least occupied nodes with data that is considered consistent or complete [18]. The data that gets written will eventually be copied over to the adjacent nodes until all of them have the same data [14].

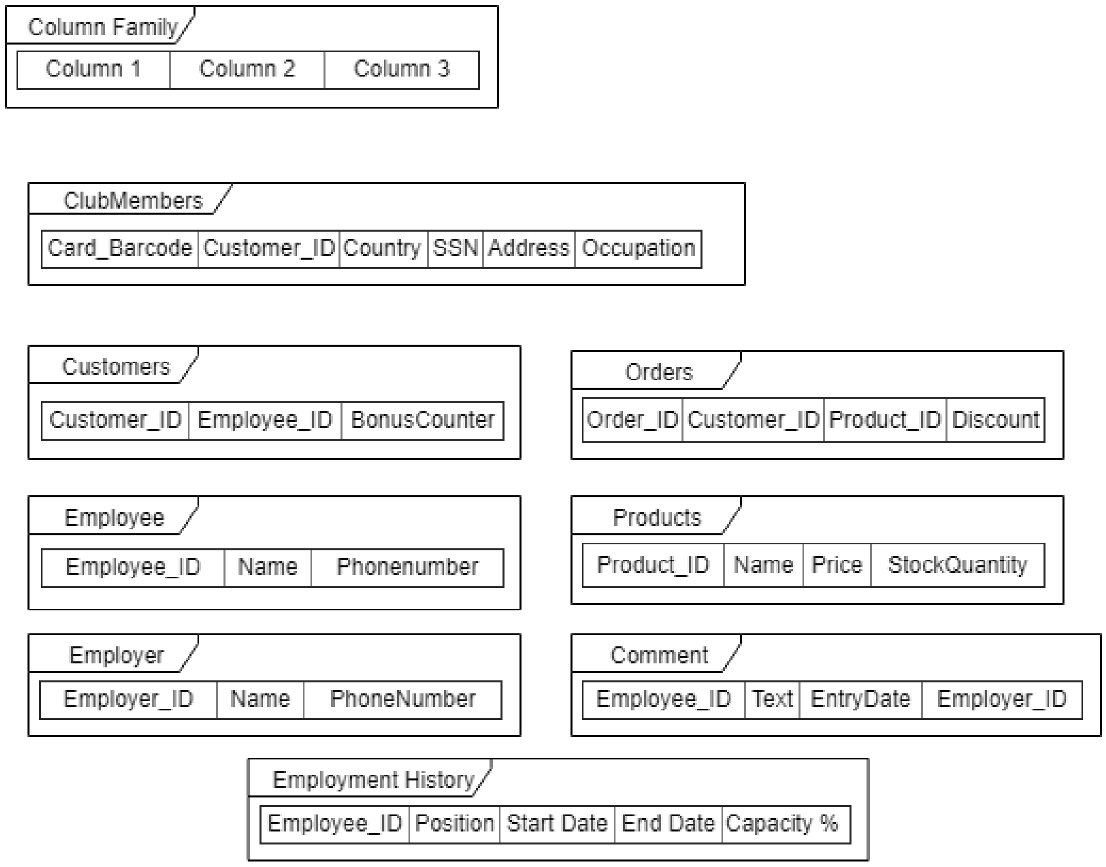

Fig. 6. Columnar-based database diagram

Key-value databases can split up the data into buckets which in relational database terms is equivalent to a table. In a standard configuration you could then just add any type of data you want into a bucket, as long as you provide it with a unique key [17]. However, with RiakKV there is also the built-in utility called Riak Data Types that could be used for buckets that contain a standardized set of attributes for each key-value entry. Normally a bucket holds a specific category of data, such as Customers, which is why they are sometimes called domain buckets [14].

RiakKV databases can also be scaled easily by splitting up the data inside the database by sharding the data, which splits the data up based on a specific attribute of the key [14]. For example, a node could store all the keys within a certain range or with a certain string. Sharding increases the performance of the database but also the complexity, requiring the user to configure the sharding nodes correctly to ensure their write/read consistency needs.

RiakKV's database model could be applied to certain components from the BeaverCoffee's requirements, with an example of a component which makes a good fit for a key-value database being the commerce aspect [14]. But Beaver-Coffee's requirements seeks to create an entire system for their business, which includes additional components such as managing employees, employee records, customers, club members, orders and product inventory.

This means the database needs to be able to handle multi-operation transactions of data. Which could happen when saving multiple keys and one or more fails to save the data, which could require a rollback or revert the operation, something that key-value stores are not the best solution for. Furthermore, RiakKV was not designed to handle correlations or relations within its data, which is one of the requirements for BeaverCoffee, for example when we need to see if a customer is also an employee so that a discount can be applied. [14]

Since RiakKV falls short on the aforementioned aspects, it is unfit for BeaverCoffee's requirements as it was not designed with those aspects in mind.

However, despite not being the best fit for the use case that BeaverCoffee requires, it is still possible to achieve this. A proposed key-value based database model diagram for the BeaverCoffee system can be found in figure 7.

IV. Conclusion

Ever since the traditional relational database came into existence it has generally been the default choice for company applications all around the world. Still widely used and well established, providing the ideal solution for many problem domains, the relational model is sometimes at a disadvantage. Requiring pre-defined schemas makes the relational model

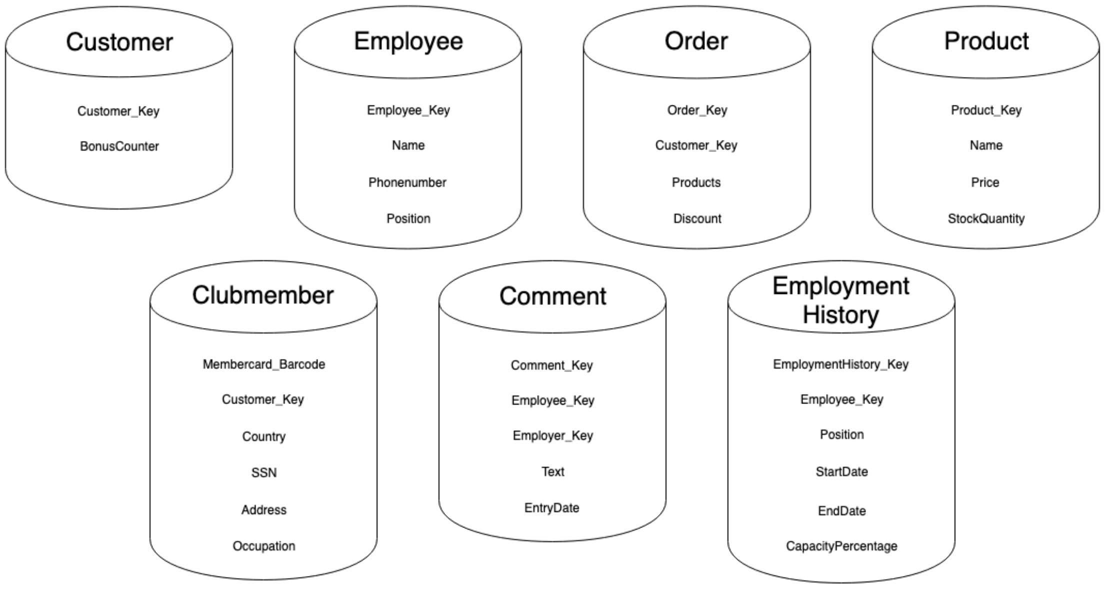

Fig. 7. Key-value based database diagram

rigid in its structure and tardy to modify. It is also unsupportive of clustering, meaning that it scales up instead of out, which makes it an expensive solution for systems generating a lot of data. For such reasons, NoSQL is the better solution for the BeaverCoffee system [2]. In opposite to the relational model, NoSQL data stores are flexible and designed to handle a variety of large, semi-structured and unstructured data. Their model and schema are based on queries to be made application-side and are designed around the question ”What questions do I have?”. In contrast, the schema of relational databases are driven by the structure of available data, to support the question ”What answers do I have?”.

For a company like BeaverCoffee, which started as a small local business but is now rapidly growing in popularity and preparing for expansion, a non-relational model is by far the better solution. Since it is likely that the BeaverCoffee company will keep growing, it is designed with the possibility of future expansions in mind. The NoSQL model provides the flexibility that is required and allows the database structure to easily be updated and adjusted to fit the company needs. It also provides much cheaper scaling due to the fact that they support clustering and scales out over cheap commodity machines. Effectively avoiding the high cost associated with traditional vertical scaling.

After careful evaluation, weighing the advantages and disadvantages of each of the four main NoSQL types, a document database was concluded to be the best implementation for the BeaverCoffee system. MongoDB is the most established and advanced out of all modern document stores and offers a rich query language, secondary indexing, automatic object mapping, aggregation features, atomic transactions and much more. In MongoDB, documents can be modeled with subdocuments and/or arrays nested inside, keeping related data neatly organized and easily accessible. For the BeaverCoffee system the documents were modeled in a way that best supported the read and write operations expected to be performed. With this design, most data needed for application functionality is easily accessed and retrieved in a single query. Write operations performed on a single document is atomic in MongoDB, meaning that as data is being altered in a write operation, the changes will not be made available until the whole document has been successfully updated (including changes made to embedded documents and nested arrays). Any errors during the write will cause the operation to roll back, effectively terminating the process and the changes made. This write isolation guarantees that all reads are consistent and that a partially updated document will never be seen. Resulting in a system with strong consistency [19].

The database solution for BeaverCoffee system has to support different application components such as managing employees, customers, product inventory, order processing and record keeping. That being said, one could safely assume that priority of CAP standards will differ between components. For example, in order processing AP might be key, whilst product inventory prioritizes CP. MongoDB is by default CP but offers the option to specify read and write concerns for the database, basically allowing the developer to regulate the level of consistency versus availability. A feature undoubtedly beneficial when implemented correctly [20].

A single database implementation was selected to support all the different components and requirements of the BeaverCoffee system. Although the solution is solid and designed to handle the different problem domains efficiently on its own, one could argue that a solution based on the idea of polyglot persistence could also have been beneficial. Polyglot persistence describes the fact that a system rarely deals with a single problem, and that multiple data stores should be implemented to deal with issues they are specifically designed to. For example, order placement could be handled by a key-value store, efficiently retrieving the required item by primary key. Also, if BeaverCoffee is expanded to offer web services to customers, a graph store could handle connected data such as product recommendations based on current and/or previous orders as well as products purchased by others. This way, data would be stored and processed in the way most suitable for each particular component, increasing the overall performance of the system. This is indeed a very plausible solution in theory, but is not considered necessary as the specified requirements are successfully achieved by the proposed MongoDB implementation.

In summary, the suggested data model and implementation will efficiently handle the requirements of BeaverCoffee system. It provides cheap scaling and a flexible structure, as well as keeping data strongly consistent throughout the database, providing high availability and partition tolerance. Additionally, it supports the possibility of expanding the system to offer future web based services. Whilst larger modifications would have to be done application-side, the underlying schema of the database would not require any substantial changes.

Individuals Contribution Statement

Betty Brändström contributed in Introduction, Description, MongoDB analysis, Conclusion, design of the implemented data model/schema, aggregate-model diagram, sequence diagram, document-model diagram and implementation of the view comments tab in the system.

Casper Strand contributed in Abstract, RiakKV portions of this document, ER diagram for the relational schema, diagram for Key-Value based databases, installation instructions in Appendix, adding launch arguments to the application (so the end user can easily build the database and add test data).

Lars Korduner contributed in Graph store database modeling, proofreading, planning and report creation in application.

Lily Hammerskog contributed in Description, Cassandra analysis, column-model diagram and implementation of currency in the system.

Mattias Sundquist contributed in MongoDB analysis, design of the implemented data model/schema, design of the application GUI, system design and implementation.

Appendix

A. Installation Instructions

The application was uploaded alongside this application to Canvas. The application expects a MongoDB Database Server on the local host, which means the user is required to download and install MongoDB Community edition with default settings. For ease of use it is recommended to also install MongoDB Compass when prompted to do so in the installer, which enables the user to easily monitor the data within the database.

When there is a MongoDB database server running the user can proceed by building all the database collections, which can be done by clicking ”_Build Database.bat” which will setup the collections as well as add some test data. Confirm that these are in place by logging on to the database using MongoDB Compass. There should be a BeaverCoffee database with collections for ”Customer”, ”Employee” and ”ItemStock”.

Finally the application can be used by clicking ”_Launch BeaverCoffee.bat”.

B. User Manual

The features and functionality available to the user is based on their employment status. Current access levels are;

- Regular employee. First/lowest level of access is given to the regular employee. The employee can place orders and add customers to the system. To access the system as an employee login with name ’Betty Brändström’ and password ’admin’.
- Location manager. As well as having access to features that the regular employee has, a location manager can create, read and update employee and customer data as well as stock quantities. They can also add individual comments to each employee. To access the system as a location manager login with name ’Mattias Sundquist’ and password ’admin’.
- Cooperate sales manager. As well as having the access to features of a regular employee and location manager, a cooperate sales manager can also create the following reports; sales over a specified period, sales for products over a specified period, sales per registered customer per address or occupation, stock quantities over a specified time period, orders served by an employee over a specified time period, employees and customer listing. To access the system as a cooperate sales manager login with name ’admin’ and password ’admin’.

References

[1] MongoDB. ”What Is A Non Relational Database”. [Online]. Available: https://www.mongodb.com/scale/what-is-a-non-relational-database [Accessed: 2019-04-22].

[2] Pramod J. Sadalage and Martin Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison - Wesley Professional, 2012. Chapter 1, Part 1.4-1.6.

[3] Pramod J. Sadalage and Martin Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison - Wesley Professional, 2012. Chapter 8, Part 8.1, 8.2.4.

[4] Pramod J. Sadalage and Martin Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison - Wesley Professional, 2012. Chapter 9, Part 9.0.

[5] Database.Guide. ”What is a Column Store Database?”. [Online]. Available: https://database.guide/what-is-a-column-store-database/ [Accessed: 2019-04-23].

[6] Pramod J. Sadalage and Martin Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison - Wesley Professional, 2012. Chapter 10, Part 10.3.

[7] Neo4J. ”What is a Graph Database?”. [Online]. Available: https://neo4j.com/developer/graph-database/ [Accessed: 2019-04.23].

[8] Pramod J. Sadalage and Martin Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison - Wesley Professional, 2012. Chapter 11, Part 11.1, 11.3.

[9] LucidChart. ”Entity-Relationship Diagram Symbols and Notation”. [Online]. Available: https://www.lucidchart.com/pages/ER-diagram-symbols-and-meaning [Accessed: 2019-05-08].

[10] MongoDB. ”NoSQL Databases Explained”. [Online]. Available: https://www.mongodb.com/nosql-explained [Accessed: 2019-05-08].

[11] MongoDB. ”Database Schema Example”. [Online]. Available: https://www.mongodb.com/scale/database-schema-example [Accessed: 2019-05-05].

[12] Robinson, Ian and Webber, Jim and Eifrem, Emil. *Graph databases: new opportunities for connected data*. O’Reilly Media, Inc., 2015. .

[13] MongoDB. ”Data Model Design”. [Online]. Available: https://docs.mongodb.com/manual/core/data-model-design [Accessed: 2019-05-05].

[14] Pramod J. Sadalage and Martin Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison - Wesley Professional, 2012. Chapter 8, Part 8.2, 8.3, 8.4, 8.4.

[15] Riak. ”Use Cases For Riak KV”. [Online]. https://docs.riak.com/riak/kv/2.2.3/learn/use-cases/ [Accessed: 2019-05-16].

[16] Srini Penchikala. ”Riak NoSQL Database: Use Cases and Best Practices”. [Online]. https://www.infoq.com/news/2011/12/andy-gross-riak-database [Accessed: 2019-05-16].

[17] Riak. ”Data Modeling with Riak”. [Online]. https://riak.com/posts/technical/data-modeling-with-riak/ [Accessed: 2019-05-16].

[18] Sargun Dhillon. ”Benchmarking Riak - Performance under Pressure”. [Online]. https://medium.com/@mustwin/benchmarking-riak-bfee93493419 [Accessed: 2019-05-16].

[19] MongoDB. ”MongoDB and MySQL Compared”. [Online]. Available: https://www.mongodb.com/compare/mongodb-mysql [Accessed: 2019-05-15].

[20] MongoDB. ”Read Concern — MongoDB Manual”. [Online]. Available: https://docs.mongodb.com/manual/reference/read-concern/read-concern-levels [Accessed: 2019-05-16].