A COMPARISON OF MONGODB, CASSANDRA, AND THE RELATIONAL PARADIGM

Carolyn Atterbury
Introduction to Databases, CS 564
Department of Computer Science
University of New Mexico

May 8, 2019

ABSTRACT

Though relational databases have become a tried and true way of modeling and storing data, the onset of big data from web-related companies like Facebook and Google have brought about a new wave of NoSQL databases. NoSQL databases differ from the relational model, in that the data stored may be unstructured. NoSQL databases are built to scale horizontally, which make them a good candidate for any company that may need to scale quickly. Though there are many popular NoSQL databases that are being used today, we will look closer at MongoDB and Apache Cassandra. By comparing MongoDB and Cassandra, we will be able to see two different approaches to storing data when immediate consistency is no longer the top priority.

1 Introduction

Relational Databases have dominated the industry since they were proposed by E. F. Codd in 1970. In recent years, however, the rise of the Web 2.0 has brought about the need for a type of database that handles large amounts of unstructured data with fast access times. NoSQL databases, which stands for "Not Only SQL", have allowed tech companies like Google and Facebook to have horizontal scaling and increased speed for most operations.

In this paper we will be looking at some of the features that differentiate NoSQL databases from the relational model, and how each model has a different approach to the CAP theorem. In addition, we will be comparing two popular NoSQL databases: MongoDB and Apache Cassandra.

2 Background

Though many companies are benefiting from the Relational Model that has been dominant in the last 50 years, the large amounts of unstructured data that is currently being generated from web-related activities has inspired the NoSQL model. The strict schema required for Relational Databases does not work well with data that does not have a guaranteed form. Additionally, the amount of data being generated from web-related activity is bigger than ever before. Companies needed a database that scales well with their systems, while maintaining the same amount of performance.

As data volume increases, and more storage is needed in the database, it becomes cheaper to add more machines to a network (horizontal scaling) than it is to add more CPUs to a machine (vertical scaling). In the case where horizontal scaling is necessary, NoSQL databases became the best database option. The NoSQL BASE properties work better on horizontally scaled systems, while the ACID properties of relational databases run into difficulties when working across multiple machines. The ACID properties, discussed in 2.1, and the BASE properties described in 2.2, are based off of the CAP theorem.

In year 2000, Eric Brewer came up with the CAP theorem that describes three principles to describe a distributed system:

1. **Consistency** - Data must be consistent across multiple servers in a system.
2. **Availability** - Every request in the system must have a response.
3. **Partition Tolerance** - The system must be robust enough to continue working when there is a partition failure.

The CAP theorem ensures that it is not possible for a single database system to have all three of these properties at all times. For example, in the case of a network partition failure, one must choose between consistency and availability. From the CAP theorem, when faced with a trade-off between consistency and availability, relational databases choose consistence, while NoSQL databases choose availability. In this section, we will give an overview of the Relational and NoSQL database model.

2.1 Relational Databases

Relational databases represent a way to store data that is based off of Relational Algebra. A "relation" in the relational model, is a table with rows and columns. The columns in the relational model represent the various attributes, or information about some object. A row is a collection of attributes that make up the object. A relation is a set of object with shared attributes. In the relational model, each row has a primary key attribute (or set of attributes) that are the unique identifier for that row. Each row in the table must be unique, and primary key attributes ensure that property. Foreign key attributes are an non-primary attribute on a row that has the same value as a primary key on another row. Foreign key attributes can be used to join separate tables, so the user can get data from both tables in one query. The action of linking separate tables through a foreign key is called a "join" operation. Join operations can be an expensive operation, as it requires all rows with the specified foreign key attribute to be matched up with the corresponding primary key attribute in another table. Any number of joins can be used in one query, but the more joins there are in a query, the longer the query takes to complete.

Before creating a Relational Database, the user must first create a relational schema, which specifies all of the relations, and all of the attributes associated to that relation. For each attribute, the user must specify the type, if it is required or optional, and if it is a primary key, foreign key, or neither. A common practice in the relational model is to Normalize the data, which saves space and prevents inconsistencies that come with data-duplication. While data normalization ensures data integrity by preventing duplicate data, it requires the user to use more joins when accessing data. When the amount of data in a relational database increases, there can be noticeable performance problems, especially when doing queries that involve joins.

Relational databases use SQL to access the data, which stands for "Structured Query Language". SQL is a declarative programming language based on relational algebra. It includes aggregate functions like SUM(), COUNT(), MAX(), MIN(), and the ability to do nested queries.

Building off of the CAP theorem, relational databases have their own set of properties, ACID, which stands for Atomicity, Consistency, Isolation, and Durability.

- **Atomicity** - Each transaction either succeeds completely, or fails completely, as a single unit. If a transaction fails in the middle of a process, the entire process is rolled back.
- **Consistency** - Each transaction brings the database into another valid state. To be in a valid state, all database invariants, and constraints, must be in accordance with the predefined rules.
- **Isolation** - Even if transactions occur concurrently, Isolation ensures that concurrent transactions do not interfere with each other. Each concurrent transaction must behave the same is if the transactions occurred sequentially.
- **Durability** - In the case of a system failure, durability ensures that a transaction will remain committed. Transaction commits are generally written to a log file that is stored in non-volatile memory.

To ensure all of these properties hold, relational databases use locking, which ensures that no transaction can edit the same data as another transaction at the same time. While the ACID properties ensure consistency across data, it becomes harder to ensure all of the ACID properties as the database gets larger. As soon as the database becomes distributed across multiple nodes, then more complications arise due to network failures. If a write operation succeeds on one node,

but fails on another node, the whole transaction must be rolled back. The two-phase commit protocol ensures atomicity in a distributed system by coordinating a consensus across nodes. In the two-phase commit protocol, a transaction is committed only if it succeeds on all nodes. While it works well in distributed systems, the two-phase commit protocol is not resilient across all types of failures. Overall, it is difficult to transfer a system with ACID properties across multiple nodes, and there is a significant decrease in performance.

2.2 NoSQL Databases

As social media companies rapidly grew in popularity, they needed to store large amounts of data, including media data, without affecting the performance of queries. While relational databases prioritize consistency, NoSQL databases choose to have easily available data which result in lower performance costs, even across multiple storage units. To achieve data availability and high performance queries, NoSQL databases tend to denormalize data. If data consistency has less priority than availability, then data duplication no longer becomes an issue, as long as the database can eventually become consistent. By keeping data denormalized, there is no longer a need to look in multiple tables to resolve foreign keys for a single query.

There are many types of NoSQL databases, but four main categories of NoSQL databases are the following:

- **Key-Value Store** - In this model, data is stored as a set of key, value pairs, where each key appears at most once in the collection.
- **Document Store** - Document store databases maintain a set of key, value pairs, that are eventually stored in a document of a certain format: JSON, or XML. Each document is associated to a unique key.
- **Column Family** - Similar to the relational model, Column Family NoSQL databases group data together in columns, where a column represents a key, value pair. Super-columns and column families are used to structure the database.
- **Graph Database** - Graph Databases structure data similar to a graph data-structure, with nodes and edges. Graph databases are useful for representing the relationships between nodes.

Each type of NoSQL database can have its own query language. Since many NoSQL databases store data in a JSON format, the corresponding query language will follow a similar JSON syntax. Other query languages, like CQL used by Cassandra, closely resemble SQL used by relational databases.

NoSQl databases have their own set of BASE principles for database transactions, which stand for Basically Available, Soft State, Eventual Consistency.

- **Basically Available** - The data is distributed across multiple nodes. If there is a failure in one node, the system will continue to work. In this way, the system ensures availability.
- **Soft State** - Indicates that, due to the eventual consistency model, the state of the database between transactions may change over time.
- **Eventual Consistency** - Indicates that the system will be comes consistent over time. A change to the database may not be consistent immediately, but the change will eventually propagate to all nodes.

The BASE properties allows the system to be more flexible than the ACID properties. This flexibility ultimately ends up increasing the performance speeds as the system scales.

3 Overview of MongoDB and Cassandra

In this section we will be comparing two popular NoSQL database systems: MongoDB and Cassandra. MongoDB is a Document Store NoSQL database while Cassandra is a Column Family database. MongoDB is written in C, but supports many languages including: Actionscript, C, C#, C, Clojure, ColdFusion, D, Dart, Delphi, Erlang, Go, Groovy, Haskell, Java, JavaScript, Lisp, Lua, MatLab, Perl, PHP, PowerShell, Prolog, Python, R, Ruby, Scala, and Smalltalk. Cassandra is written in Java.

3.1 MongoDB

MongoDB is a Document Store NoSQL database written in C++. It is an Open Source project and the data is stored in BSON documents with a maximum size of 16 GB. BSON stands for Binary JSON format. The document is structured in key, value pairs, where a field can be any of the BSON data types. The BSON data types include primary types as well as a timestamp type, date, arrays, other documents, and arrays of other documents.

3.1.1 Modeling Data

MongoDB has collections, which is a set of documents that share a similar structure, although they allow for completely unstructured data as well. For efficiency, they recommend that users incorporate structures into their data, using schemas. Collections in MongoDB are similar to Relations in a relational database, and a schema can be created for the various collections in the database, although all documents in the schema are not required to have the same schema. The set of fields across documents can be different in a collection. Additionally, the types associated with a field can differ across documents in a collection. MongoDB allows for embedded data, by storing documents and arrays, in documents. Though it is possible to have normalized data by storing foreign keys in documents, MongoDB recommends denormalizing data by taking advantage of embedded data structures.

Each collection is assigned an immutable UUID, a Universally Unique Identifier. This UUID remains the same for that collection across all shards in a sharded cluster, and all members of a replica set. Additionally, each document in a collection is required to have an unique identifier that serves as the primary key of that document for that collection. If not specified on a document in an insert operation, the unique identifier field "_id" is automatically assigned a value which is a MongoDB ObjectId. The ObjectId creates a unique Id value by including a timestamp of when it was generated, along with a random value and a counter.

3.1.2 Indexing

To improve performance, MondgoDB uses indexing similar to relational databases. An automatic index is created for each collection based on the "_id", but the user can specify any number of indexes for a given collection, and any number of fields on a given index. Indexes can dramatically improve the time involved in read operations as each of the fields in the index are sorted. If the read query is only asking for fields that exist in the index, then the values in the index are returned without having to access the full document in memory or on disk. Insert and Update operations can be slower, however, as the data modification must be applied to all indexes related to the document in addition to the document itself.

3.1.3 Building Queries

MongoDB supports CRUD operations: Create, Read, Update, Delete, as well as a bulk write operation. Create (insert) operations can be used to add a document to a collection. If the collection does not exist, then the insert operation will create a new collection and will add the document to that collection. Every write operation in MongoDB is atomic with respect to that document. It is possible for a user to insert or update many documents within a single transaction. In this case, each document is modified in an atomic way. While it is possible to update many documents in a single query, multi-document transactions are slower than single write operations.

There are multiple approaches that MongoDB has towards using aggregation operations. Aggregation operations generally look at multiple documents and eventually return a single result. MongoDB performs aggregation using the Aggregation Pipeline, the map-reduce function, and single purpose aggregation methods.

- **Aggregation Pipeline** - Documents enter a multi-stage pipeline that can either filter or modify the output document, or it can sort or group the output.
- **Map-Reduce function** - A Map phase iterates over each document, and can return one or more documents each iteration. The reduce phase takes all the output from the map phase and modifies it.
- **Single Purpose Aggregation methods** - Can be used to aggregate documents from a single collection, using built-in count() and distinct() functions.

3.1.4 Maintaining Durability and Availability

MongoDB maintains durability through using replica sets, which is a group of processes that maintain the same data. Replica sets improve data availability through adding redundancy in the data. This also provides improved fault tolerance, as the data is duplicated across multiple servers. Each replica set has one primary node, while the other nodes are considered secondary nodes. The primary node receives all read and write operations and records all changes in the operation log. The secondary nodes look in the operation log for changes, and then update their database to reflect the changes. If the primary node fails, a secondary node will be promoted to become the new primary node. This approach to replica sets follows the Master-Slave model, where the primary node is considered the Master node. There can only be one Master node in each replica set.

In addition to having replica sets, MongoDB uses sharding to distribute data across multiple machines. Sharding allows the database to distribute the workload of read and write operations across multiple machines. In this case, each shard would process a subset of the transactions. Sharding also distributes the storage capacity over multiple machines. If one or more of the shards becomes unavailable, then transactions can still take place on other shards, making the data highly available.

3.2 Cassandra

Apache Cassandra is an open source NoSQL database that uses the Column Family structure to store data. Using tables with columns and rows of data, Cassandra database is similar to the relational model, but instead of grouping data according to entities, Cassandra creates tables based on likely queries. The data stored in a Cassandra database can be structured, semi-structured, or unstructured.

3.2.1 Modelling Data

In the relational model, the database design process starts with a conceptual understanding of the entities involved in the database and how they relate to each other. Queries are often not part of the database design process with this object oriented approach. Cassandra has a different approach. When coming up with a model for a Cassandra database, the user starts with the queries and creates tables by grouping attributes together that will likely be requested in a given query. The data stored in tables is denormalized, so one query can return the necessary information. Cassandra does not allow joins or subqueries, so denormalization and data duplication is necessary. Each table has a partition key, and columns are grouped together in column families, where each column can have a specific sort order.

3.2.2 Queries and Transactions

Since the structure of Cassandra data is similar to the relational model in that there are tables with columns and rows, Cassandra's query language CQL borrows a lot of the syntax from the relational query language SQL. CQL has a set of identifiers and keywords that can be used to identify tables and columns. CQL has many data types including: timestamp, UUID, counter, duration, inet, map, set, list, and others. All CRUD operations can be used in Cassandra as SELECT, INSERT, UPDATE, and DELETE statements. Additionally, a BATCH statement can be used to group multiple modifications into a single statement.

Modifications in a BATCH statement all occur in isolation, and all the modifications must be completed in entirety, or the BATCH operation will roll back. When doing BATCH writes to the database, Cassandra uses row level atomicity, and row-level updates are made in isolation.

Aside from having built-in aggregate functions (COUNT, SUM, MAX, MIN, and AVG), Cassandra allows for user-defined aggregate functions that can be used in a SELECT statement.

Cassandra allows adding multiple indexes to tables, in addition to adding indexes to fields with a map type. Each node then keeps track of the indexes for the tables in that node, and the indexes are stored as another type of table in that node. While indexing can speed up some queries, it can also add overhead which can slow down transactions.

3.2.3 Durability and Availability

To ensure durability and availability, Cassandra allows the creation of replica tables with two replication strategies:

- **Simple Strategy** - Relies on a user defined "replication_factor" that determines how many nodes should contain a copy of the each table. All nodes are treated equally, and they are all have a set of replicas equal to the "replication_factor".
- **Network Topology Strategy** - Allows a "replication_factor" to be configured for each datacenter in a cluster, where a datacenter is a group of related nodes.

The user can choose whether to use synchronous, or asynchronous replication. Since all the replication sets respond to read and write requests equally, Cassandra is known to follow the Many Masters approach to network requests.

Cassandra also allows the user to modify the consistency parameters for each operation in the database. Using configurable Consistency Levels, the user can specify how many replicas need to respond to a request before the transaction is considered successful. In this model, all replicas are sent the request, but only the configured number have to respond. Though consistency parameters are customizable, they do recommend certain setting to guarantee strong consistency. Suppose N is the number of replica nodes, W is the number of replicas that respond to an update request, and R is the number of replica nodes that are contacted when data is accessed through a read request. Then, in order to maintain strong consistency, $W + R > N$. In this case there is an overlap between the set of nodes responding to write requests and the set of nodes responding to read requests. If $W + R \leq N$, then the read and write set may not overlap, and there could be weak consistency in the system.

Each transaction in Cassandra is written to a CommitLog, which keeps track of all mutations that happen on a Cassandra node. After writing to the CommitLog, the transaction writes to a memtable, which is a structure in memory where Cassandra buffers write operations. Eventually the memtables become SStables, which is immutable data that is stored on disk. If the node unexpectedly shuts down, then the transaction history will be preserved for that node on the CommitLog. As soon as the system starts up again, unfinished transactions that were added to the CommitLog get added to the memtable, and eventually become an SStable. This procedure improves durability, as the transaction will still finish even if there was a shutdown.

4 Comparing MongoDB and Cassandra

Though there are many similarities between MongoDB and Cassandra, there are many differences as well. The most notable difference between the two is the different priorities in responding the the CAP theorem. By prioritizing Consistency and Partition Tolerance, MongoDB picks the CP type system in CAP. This is noticeable in their choice of the Master-Slave model in the data replication sets. Cassandra chooses a AP type system, meaning that they prioritize availability and partition tolerance. Using the Many Masters model with their replication sets, the Cassandra system will always return a response immediately from a request, even if a node goes down.

4.1 Data Modeling

In Cassandra, a query-first approach is taken when coming up with a way to model the data. The tables in Cassandra are structured according to the queries being used, and columns are grouped together to ensure that each query can be fulfilled without any table joins. Though MongoDB is flexible enough to have any type of data model, they do encourage structuring collections using a schema with an approach similar to the relational model. In this way, MongoDB has more of an object oriented approach to data storage. While it is possible to have 3CNF normalized data in MongoDB, they encourage users to take advantage of nested documents while denormalizing the data.

Both Cassandra and MongoDB have triggers which can be used to keep denormalized and duplicate data in sync, and they both allow for additional indexing to improve performance on queries. In MongoDB, a user can create an index on any property in a document, including on nested documents. Cassandra only allows secondary indexes on single columns.

4.2 Data Storage

Data, in Cassandra, is stored in tables on various nodes in the cluster. A table in Cassandra consists of columns and rows. A user queries a specific table to retrieve the appropriate data from that table. Indexes in Cassandra are stored in a different type of table on the same node as the table it is an index for. First a change is recorded in the changeLog, then it is reflected in a temporary memtable, before it is added to disk storage in an SStable.

In MongoDB, data is stored in BSON documents and grouped together using a UUID that corresponds with the collection it belongs to. Documents in a collection can be distributed across many machines in a cluster.

4.3 Durability and Availability

Both MongoDB and Cassandra have replica sets that copy the data and are available in case a node goes down. MongoDB has a primary node and secondary nodes in a replica set, which follows the Master-Slave model. If a primary node goes down, one of the secondary nodes gets promoted to primary. If this happens, the response time on write queries could significantly slow down until a new primary node is chosen. Cassandra has a Multiple Masters model, which means that the system will not lose any performance if a node goes down.

Due to the Master-Slave model in MongoDB, since the primary node is the only node to process write operations, then MongoDB will not be able to handle as many write operations as Cassandra. In Cassandra, a write operation can be processed on any node.

4.4 Querying

Cassnadra and MongoDB have very different query languages, although ultimately they share the same ability to perform CRUD operations, including with a bulk option. Cassandra's query langugage CQL uses the same syntax as SQL, and may be easier for people transitioning from relational databases. CQL does have limitations, in that no JOIN operations are allowed, and it does not support the operation "OR". MongoDB uses a JSON style query language which mimics the BSON structure that the data is stored in.

MongoDB and Cassandra both support aggregate functions, although MongoDB has more infrastructure for customizing complex function with the aggregation pipeline.

4.5 Performance

Without doing explicit performance benchmarks, it is hard to tell which system will have better performance. Simply based on the difference in their data replication models, where MongoDB has a Master-Slave model and Cassandra has the Many Masters model, it is likely that Cassandra has higher performance on write operations, since every node in the system can perform write operations. This is unlike MongoDB, where only primary nodes can process write operations. As every node in MongoDB can process read operations, then MongoDB and Cassandra should have the same expected performance for read operations.

When comparing performance benchmarks between MongoDB and Cassandra, Ambramova and Bernardino[1] found that in over 8 different benchmark tests, Cassandra overall had better performance at increased data sizes. They noticed that MongoDB was slightly slower than Cassandra when handling large numbers of insert operations, which is to be expected with the Master-Slave model.

It should be noted that the performance in each system depends on how the data is structured, and the replication parameters that are in place. Since MongoDB and Cassandra have such different ways of structuring the data (a more object oriented approach, versus a query-based approach), it may be hard to do an exact comparison. Users should be careful to structure the database according to the needs of each system in order to maximize performance on that platform.

5 Conclusion

We started this paper by discussing the Relational model and some of the shortcomings that gave rise to NoSQL databases. As companies began gathering large amounts of data, they needed to scale quickly and in a way that was the least expensive. Since horizontal scaling was the easier and cheaper alternative to scaling, companies needed a database that continued to work well with high performance across multiple machines. By loosening up some of the constraints by allowing for eventual consistency, NoSQL databases are able to distribute data across multiple machines while maintaining a system that continues to function even if there are nodes that stop responding. The table below illustrates some of the main differences between Relational and NoSQL databases.

Within the NoSQL approach, there are many different approaches to data storage. After taking a closer look at MongoDB and Cassandra, it is apparent that each database has different priorities when approaching the CAP theorem.

Table 1: Comparing Relational Databases against NoSQL Databases

	Relational Model	NoSQL
CAP Theorem	Consistency and Availability	Parition Tolerance, and either Availability or Consistency
Transaction Properties	ACID	BASE
Horizontal Scaling	No	Yes
Structured Data	Yes	Not Necessarily

Table 1 compares the main differentiating features between Relational databases and NoSQL databases.

By prioritizing Availability and Partition tolerance, Cassandra ends up being the faster system when dealing with large amounts of data, but it sacrifices consistency as a result. By ensuring better consistency between nodes, MongoDB loses performance for write operations, but maintains the same amount of performance for reads. Both systems are robust against system failures, which is an important feature when distributing a database across many nodes. Table 2 summarizes the key similarities and differences between MongoDB and Cassandra.

Table 2: Comparing MongoDB and Cassandra

	MongoDB	Cassandra
Storage Type	BSON	Column
Replication Type	Master-Slave	Many Master
CAP Properties	Consistency and Partition Tolerance	Availability and Partition Tolerance
Query Language	JSON	CQL
Aggregate Functions	Yes (more functionality)	Yes (less functionality)
Triggers	Yes	Yes
Atomicity	Yes	Yes
Indexing	Yes (more functionality)	Yes (less functionality)
Data Model	Object Oriented	Query-Based

Table 2 looks at some of the differentiating features between MongoDB and Cassandra.

Both MongoDB and Cassandra are popular NoSQL databases. Since it was released in 2008, Cassandra has been used by AppScale, Constant Contact, Digg, Facebook, IBM, Instagram, Spotify, Netflix, and Reddit, among others. Since it was created in 2009, MongoDB has been used by Google, UPS, Facebook, Cisco, eBay, BOSH, Adobe, SAP, Forbes, and others. Ultimately, Cassandra is the better system for scaling quickly with little overhead. Cassandra is also thought of as the better system for handling transactional data that can be found in accounting systems. MongoDB has been found to work well for content management systems, real-time analytics, and e-commerce.

For users that are familiar to the relational model, then the Cassandra query language CQL will be easy to adjust to, although the query-based data modeling approach will be different. For those who are willing to try a new query language, MongoDB may be the more familiar option, as Collections and Documents are conceptually the same as Relations and Tuples in the relational model. One could implement a normalized relational model in MongoDB, but it will not be using MongoDB in a way that is optimal for performance. Both databases are a good example of the NoSQL approach to storing data, which is becoming increasingly necessary in the current age of big data.

References

[1] ABRAMOVA, V., AND BERNARDINO, J. Nosql databases: Mongodb vs cassandra. In *Proceedings of the international C* conference on computer science and software engineering* (2013), ACM, pp. 14–22.

[2] APACHE CASSANDRA. *Apache Cassandra Documentation*. https://cassandra.apache.org/doc/latest/, 2019.

[3] BĂZĂR, C., IOSIF, C. S., ET AL. The transition from rdbms to nosql. a comparative analysis of three popular non-relational solutions: Cassandra, mongodb and couchbase. *Database Systems Journal 5*, 2 (2014), 49–59.

[4] MONGODB. *MongoDB Documentation*. https://docs.mongodb.com, 2019.

[5] PARKER, Z., POE, S., AND VRBSKY, S. V. Comparing nosql mongodb to an sql db. In *Proceedings of the 51st ACM Southeast Conference* (2013), ACM, p. 5.

[6] WANG, G., AND TANG, J. The nosql principles and basic application of cassandra model. In *2012 International Conference on Computer Science and Service System* (2012), IEEE, pp. 1332–1335.