NoSQL: an examination of Cassandra, MongoDB, and Redis

NoSQL databases are becoming increasingly popular and controversial in our day and age. It seems that there are staunch advocates and detractors of NoSQL databases. In an age where software touches almost every aspect of our lives, the correct design and implementation of software systems is of vital importance. I have decided to research and report on three NoSQL database systems: Cassandra, MongoDB, and Redis. I plan to delve into the intricacies of each one, detailing their strengths and weaknesses. I will compare each database not only to its NoSQL counterparts but also to relational database schemas.

Cassandra

History

When talking about NoSQL databases, Cassandra is one of the most popular and one of the most mentioned, especially in terms of overcoming relational database faults. Cassandra was initially developed by Facebook. Facebook is obviously a site concerned with scalability. They have billions of users and thus billions of accounts, photos, statuses, comments etc. They initially developed Cassandra to use for their inbox search feature. In July 2008 Cassandra was released as an open source project. Shortly thereafter, in March of 2009 it became an Apache Incubator project, and in February 2010 it became an Apache top-level project. Since then, various updates have been released. For example the 0.6 release in April 2010 added support for integrated caching and Apache Hadoop MapReduce. The 0.8 release in June 2011 added the Cassandra Query Language (CQL) which is very similar to SQL syntax. The latest version (2.2.2) was released on October 05 2015.

Data Model

Cassandra employs elements from both Google’s BigTable and Amazon’s Dynamo which are both popular NoSQL implementations. Cassandra uses the BigTable data model and uses architectural aspects created by Dynamo.

Cassandra stores data using the column family data model. This model is very similar to a relational schema and many parallels between the two may be drawn. However, unlike relational databases, not all rows must have the same set of columns. This makes it so that you don’t have to completely model your column schema up front. Below is a graphic illustrating the column family data model.
The most basic unit used by Cassandra is a column. A column consists of a name, a value, and a timestamp. A row is a collection of columns identified by a name. As mentioned before, not all rows are required to have the same columns. Groups of similar rows are called column families and are also labeled by a name. Groups of similar column families are called keyspaces and are also labeled by a unique name. Below is a table comparing similar aspects of relational database models and Cassandra.

<table>
<thead>
<tr>
<th>Relational Model</th>
<th>Cassandra Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database</td>
<td>Keyspace</td>
</tr>
<tr>
<td>Table</td>
<td>Column Family (CF)</td>
</tr>
<tr>
<td>Primary key</td>
<td>Row key</td>
</tr>
<tr>
<td>Column name</td>
<td>Column name/key</td>
</tr>
<tr>
<td>Column value</td>
<td>Column value</td>
</tr>
</tbody>
</table>

Using this data model, data can be added, retrieved, and updated much the same way it is in a relational database. CQL statements are similar to SQL statements. Below are some examples of valid CQL statements. While syntax is similar, it is important to remember that Cassandra works differently, internally than relational databases. Columns are not required to match up in each row. It is also important to remember that Cassandra does not support joins or subqueries.

```
SELECT * FROM MyTable;

UPDATE MyTable
SET SomeColumn = 'Some Value'
WHERE columnName = 'Something Else';

CREATE TABLE test ( Foo int PRIMARY KEY,
"Bar" int
```
Physical Storage

Data is stored physically in three locations: a memtable, commit log and SSTable. The figure below illustrates the physical storage that Cassandra uses.

When data is written it is stored in memory in the memtable. The write is also appended to the commit log. When the memtable is full, it is put on a queue to be flushed to the SSTable. As soon as the data is flushed to the SSTable, the corresponding data is taken out of the commit log. The main purpose of the commit log is to recover data from the memtable in the event of a hardware failure.

Transactions

One of Cassandra’s main selling points is that there is no single point of failure. Data is distributed across all nodes of a cluster. There is no master node, every node is the same. Fault tolerance is achieved through data replication whether it be across node clusters or even across data centers.

With this model, the main sacrifice that Cassandra makes is with consistency. The best that Cassandra can do is eventual consistency. This means that if data is written to a node, that data will eventually be available to all nodes across the cluster. However, when that data is written there may be nodes which still contain the old data.

Scalability

Another one of Cassandra’s main selling points is how well it scales. Cassandra’s architecture allows for increased read and write throughput with the addition of new machines. Cassandra scales horizontally in a linear fashion. This means that as hardware is added it can immediately be used to linearly increase read and write throughput.

Netflix performed a benchmark test to prove the scale-up linearity of Cassandra. They found that it does indeed scale linearly and they were able to do over a million writes per second. Below is a graph of their benchmark test.
MongoDB

History

MongoDB was created by a company called 10gen which would eventually rename itself MongoDB Inc. 10gen was initially built as a platform as a service (PaaS) company. The founders of 10gen did not think that any existing database model met their standards for cloud architecture. Having worked in various other companies and startups, they saw scalability as a humongous obstacle that many companies were trying to overcome. They decided to create their own document based database to use, which they called MongoDB. 10gen eventually realized that their cloud platform was not taking off but that MongoDB had a lot of potential. They decided to release MongoDB as an open source project and focus solely on developing it.

Data Model

MongoDB distinguishes itself as a document based database. Rather than storing data across tables data is stored in documents. MongoDB uses binary JSON, or BSON, to store data. A document is composed of various key-value fields. Similar records are grouped together in collections. If we were to draw parallels with relational database models, a collection would be the equivalent of a table. Each document could then be considered analogous to a row and all of the fields within a document would be analogous to columns. The following table shows these comparisons.
The document based model allows for dynamic schema creation. Collections can be created without a definitive structure. The schema for any given document can be changed by deleting or adding fields. Documents within the same collection are not required to have the same fields. However, it is common for collections to have mainly homogenous document schemas. MongoDB does not support joins or multi-document transactions. It does support a variety of atomic operations on single documents though.

MongoDB has client drivers for most popular programming languages, which in most cases are designed to be idiomatic for the given language. The document based architecture of MongoDB allows for a wide variety of complex queries. Rather than being limited to key-value operations, MongoDB supports operations such as native analytics, text search and geospatial features. The figure below illustrates the MongoDB architecture and what it makes possible.

Physical Storage

MongoDB will automatically use whatever memory is free on a given machine for its cache. MongoDB uses lots of RAM but will yield memory to other processes that request it. As needed MongoDB will use disk space. MongoDB’s “working set” is the set of data that clients access most often
and this is what is primarily kept in memory. If part of the working set must be put on disk it will significantly decrease performance due to the reads from disk.

While data in RAM is eventually written to the data file on disk, MongoDB uses a journal file to prevent data loss. When writes occur, MongoDB will record this right away on disk by appending to the journal file. So if the system crashes before data is flushed to the data file, the journal file is there as a backup.

**Transactions**

As mentioned before, MongoDB does not support joins or multi-object transactions. However, writes to a single document or any change to a single document is atomic. However, an operation that modifies multiple documents remains atomic on the document level but not on the operation level. This means that operations may interleave.

A protection against interleaving operations is the `$isolated` operator. The `$isolated` operator can prevent processes from interleaving once the write operation modifies its first document. However, isolated write operations still do not provide “all-or-nothing” atomicity. This is because if errors occur during the write operation, the write operation will not roll back all the changes that occurred before the error.

In general MongoDB will sacrifice availability in favor of consistency. MongoDB supports master-slave replication and replica sets. This means that a primary or master node is the only one capable of accepting writes. Reads automatically go the primary node but they can be sent to secondary or slave nodes as well. Secondary nodes replicate from the primary node. If a primary node fails, one of the secondary nodes will be “elected” to replace it. However, writes will be blocked until there is a functional primary node. In this manner consistency is given a higher priority than availability.

**Scalability**

The main way that MongoDB scales is through sharding. Sharding is the process of horizontal partitioning, or adding hardware to make datasets smaller. Sharding allows for the addition of new machines and thus promotes load balancing. Sharding also allows for zero points of failure and automatic failover, both of which are important to maintain consistency.

Because of the setup of MongoDB, sharding will not automatically solve scaling issues. One negative consequence of scaling horizontally through sharding is the loss of unique secondary indexes which will directly affect performance. Also, much of the performance in MongoDB will depend on schema design and implementation. If document schemas are poorly designed or grouped, then scaling will be increasingly difficult no matter what you do.

**Redis**
History

Redis was created in 2009 by an Italian programmer named Salvatore Sanfilippo. He made Redis to improve performance of a web analytics product that was part of his start up. In the same year Salvatore deployed Redis in production and completely took out their MySQL installation. Redis quickly gained attention and popularity. In March 2010 VMWare hired Salvatore to work full time on Redis. Redis is currently sponsored by Pivotal software.

Data Model

Redis is often described as a data structure server. Redis is a key-value database that works entirely in memory. Redis stands for Remote Dictionary Server. What distinguishes Redis is its ability to support a variety of abstract data types beyond strings. Most key-value databases only support strings as values. Redis however, supports lists, sets, sorted sets, hash tables, and HyperLogLogs.

Data manipulation is based off of which primitive key-value types are used. Since most programmers are familiar with these data types, this makes working with the data easier.

Physical Storage

One of the distinguishing features of Redis is it only uses memory. Redis will typically keep the whole dataset in memory. Because of this, speed is greatly increased. While Redis is mostly focused on in-memory storage and operations it does perform point-in-time snapshots of the datasets which it saves to disk.

Redis has various options and configurations in terms of how often data is saved to the disk and how it is done. It is generally accepted though that full durability is not the strong point in Redis. If hardware fails for some reason it is very likely that the most recent data written to Redis will be lost.

Transactions

Redis is not a transactional database. Unlike relational databases there is no rollback functionality when using Redis. However, some ACID properties are maintained. Atomicity is preserved both in single commands as well as group commands if they are configured correctly.

Isolation can be maintained if blocks are implemented on write functions. Data consistency is always guaranteed in Redis. In terms of the CAP theorem, Redis prefers consistency and partition tolerance over availability.

Scalability

Redis scales mainly through partitioning. By partitioning data sets into different Redis instances, more computational resources can be used and performance improves. There are, however, some setbacks to partitioning. For example, you cannot do an intersection of two data sets. Also, adding Redis instances is a lot more difficult than with other database setups. It is recommended to start with many instances to begin with and grow into them.
Differences

Cassandra, MongoDB, and Redis are all unique in the way they manage data. All of them have their strengths and weaknesses. By researching these three databases I was able to compare and contrast their capabilities and determine which use cases fit best with which system. Cassandra scales better than MongoDB or Redis. It can scale to massive sizes and maintain performance. It is extremely fault tolerant with no single point of failure. For these reasons I feel that Cassandra is especially suited for large applications with increasingly large data sets. It was built to handle large amounts of writes. It makes sense that Cassandra was developed at Facebook, considering the enormous amount of data that Facebook deals with even on a daily basis. Since Cassandra is an eventual consistency system it would not be suited for any sort of situation requiring full consistency. Also any situation requiring the join capabilities of normal relational databases would probably not lean towards the use of Cassandra.

MongoDB on the other hand has different advantages and disadvantages. The document based system provides a lot more flexibility when it comes to data schemas. Schemas do not have to be completely defined at the outset and they can be easily changed. This is a huge advantage over relational database abilities. When your data set is not fully relational, using MongoDB will give improvements in performance and scalability. An example of this is product data management. E-commerce websites can easily use MongoDB to model and manage their product catalogs.

Redis is also unique in its capabilities. As a key-value store it is extremely effective, especially because it can handle various primitive types such as lists, sets, and hashes. Redis works completely in memory so it will generally not work well on enormous data sets. The main advantage of Redis is speed and ease of access. An obvious use case for Redis would be as a cache for a web application. The set up for Redis makes it especially equipped to serve as any kind of data cache.

Conclusion

This research report was an in depth yet nowhere near exhaustive look at NoSQL databases. My examination of Cassandra, MongoDB, and Redis taught me a lot about both the why and how of using NoSQL databases. The most important thing that I learned was that there is no overall correct solution. People started using NoSQL databases to address problems they were having with relational databases, namely scaling and performance. While NoSQL databases address these issues in a variety of ways, each has their own strengths and weaknesses. Each data set will have its own unique needs and limitations. When deciding what kind of database to use it is extremely important to analyze your needs, specifications, data set, and all other relevant factors. By researching and understanding the inner workings of different databases we can make an informed decision concerning which one will suit our needs.
References


