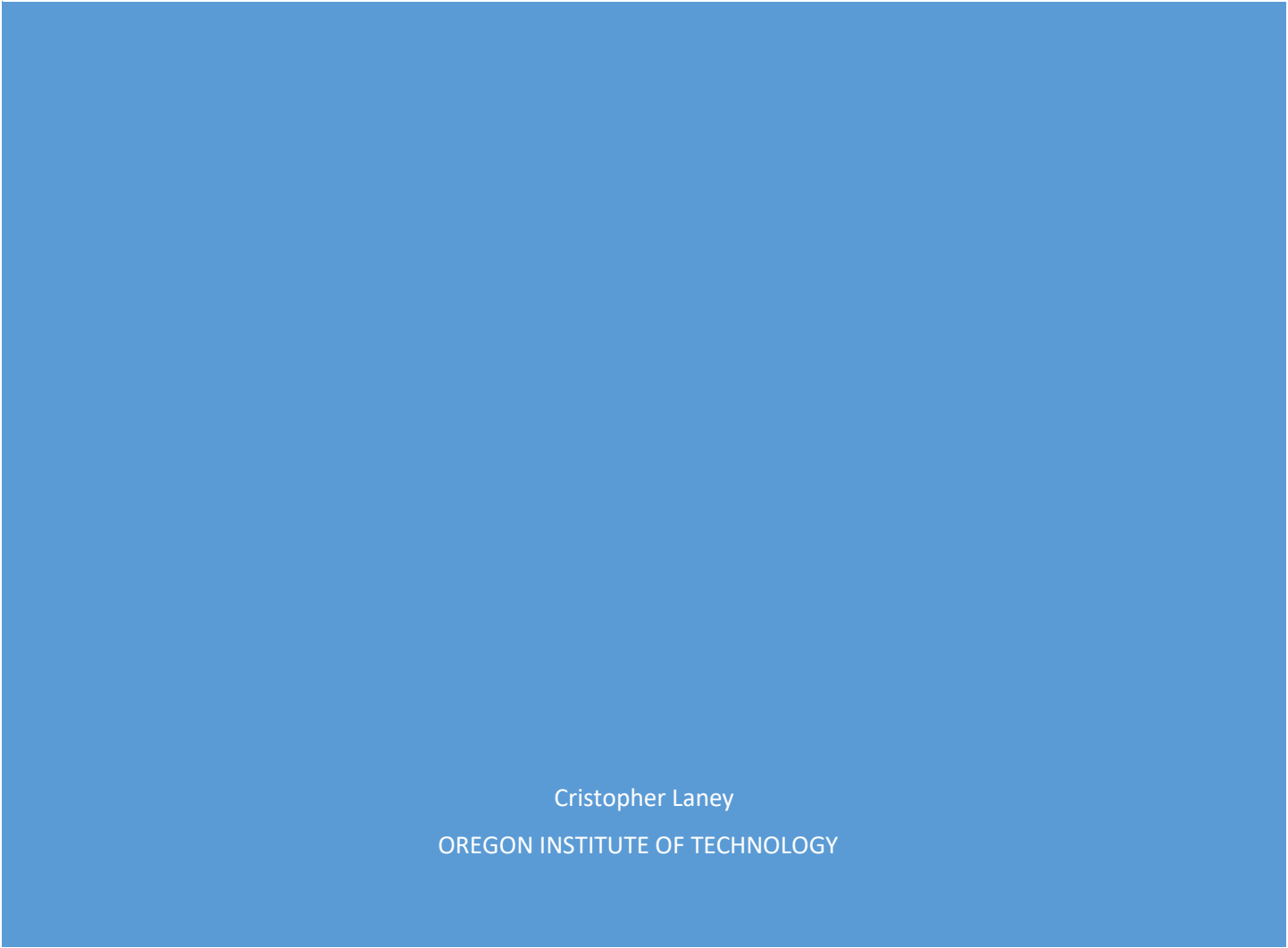




# COMPARING CLUSTERS IN SQL SERVER AND MONGODB



Cristopher Laney  
OREGON INSTITUTE OF TECHNOLOGY

## Table of Contents

Table of Figures .....	2
Introduction.....	3
Clustering in SQL Server .....	3
Synchronous Commit Mode .....	4
Asynchronous Commit Mode .....	5
Automatic Failover .....	6
Manual Failover .....	6
Forced Failover.....	7
Benefits of Failover Clustering in SQL Server .....	8
Clustering in MongoDB .....	9
Clustering for Performance (Sharding) .....	9
Clustering for availability (Replication).....	12
Foursquare .....	15
Benefits of Clustering in MongoDB .....	16
Conclusion/Recommendation .....	17
References .....	18
Figures.....	19

Table of Figures

Figure 1. Windows Server Failover Clustering Cluster ..... 4

Figure 2. Manual Failover ..... 6

Figure 3. Forced Failover..... 7

Figure 4. Hashed Sharding ..... 11

Figure 5. Ranged Sharding ..... 12

Figure 6. Replication Diagram ..... 13

## Introduction

MongoDB and SQL Server are two of the top five most popular databases. MongoDB is a No-SQL database that relies on JSON documents to store data whereas SQL Server is a more traditional Relational Database. The differences between sql, and no-sql databases are vast and far outside the scope of a fifteen-page paper. Instead, the focus of this paper will be on two particular features that both databases implement, clustering. Clustering is the grouping of data in a database in order to either increase availability or performance. SQL Server has built in methods for redundancy and availability, but lacks a built in method for clustering for targeted towards performance. MongoDB has clustering for both performance and availability, via Sharded clusters and Replication clusters respectively.

MongoDB is useful in performance critical scenarios where the database needs to scale horizontally while maintaining fast query time. MongoDB's ability to quickly and efficiently horizontally scale is why many startups choose it as their primary database. MongoDB also has a solution for high availability called replication, and replication in MongoDB acts extremely similar to clustering for high availability in SQL Server.

SQL Server is much more consistent and time proven than MongoDB, and while SQL Server does not provide functionality for performance clustering it does have a high availability cluster mode built in, involving a primary node with replicated secondary nodes, and multiple failover options.

## Clustering in SQL Server

SQL Server allows has built in methods for availability clustering, via Always On Failover Cluster Instances. These Cluster instances allow nodes to fail without the entire database being lost. The instance relies on clusters of replicated data through a primary node. If the primary node is lost, a secondary node can simply replace it, because all of the data from the primary is constantly being replicated on the secondary.

SQL Server implements a combination of nodes in a cluster to achieve high availability. Each cluster consists of one Primary node, and one or more secondary nodes. The primary node is what all read and write operations act on, unless otherwise specified. The primary is an instance of the data, and is manipulated as one would manipulate a single database. Being able to manipulate the cluster as a single database is one of the major benefits of a replication cluster in SQL Server, as it takes out a lot of manual manipulation and maintenance that would otherwise be needed. The primary node can have up to 8 secondary nodes. Secondary nodes exist to recover from a failure in the primary node. The primary node writes all data changes (Writes/updates) to a transaction log. The log is continually updated and read by the secondary nodes, and then the actions on the log are replicated in the secondary nodes, in order to keep the data up to date. When a primary node fails, a secondary will take over as the primary. Once the new primary takes over, it will roll back any uncommitted transactions.

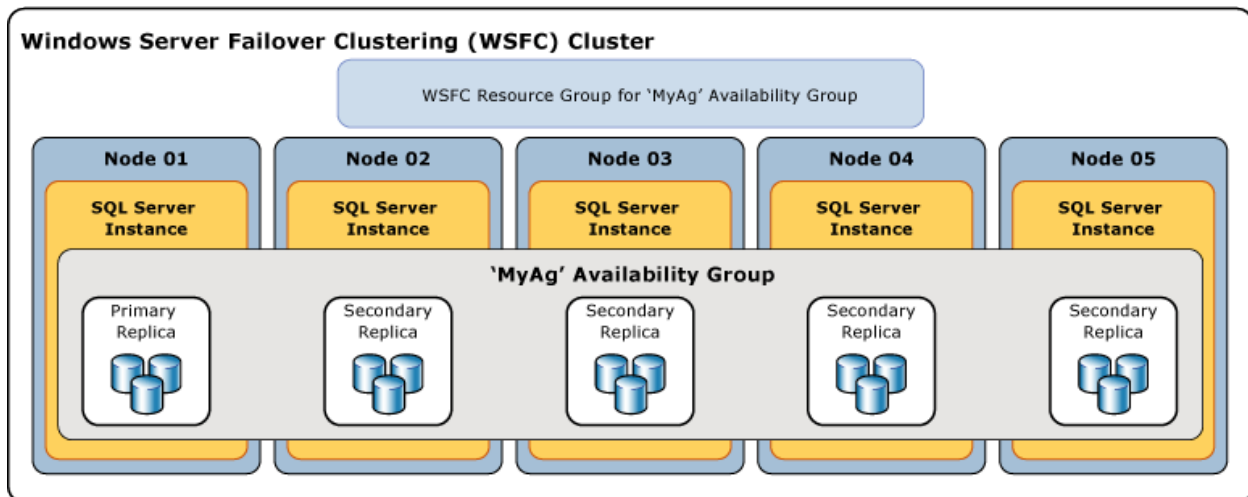


Figure 1. Windows Server Failover Clustering Cluster

Figure 1 is a graphical representation of what a Windows Server Failover Cluster looks like. The Primary Replica in Node 01 is where all reads and writes are directed. SQL Server does allow for reads to be performed directly on Secondary Replicas, but it is generally not recommended. The secondary replicas can improve read speeds in some situations, but most of the time provide little to no improvement in query speed. The transaction log will be written to for any write that occurs on the primary replica, and each of the secondary codes will cache said transaction log, and begin replicating each of those actions. Should the primary replica become unavailable, a secondary replica will step up and replace it. If the primary becomes available again, it will be added as another secondary replica which will eventually become a replica set after all changes are applied from the transaction log.

There are two availability modes for Windows Server Failover Clustering. Asynchronous-commit mode, and Synchronous-commit mode. Each of these provides different benefits and drawbacks, and should be catered to the needs of both the database and the overall project.

## Synchronous Commit Mode

Synchronous commit mode is an availability mode in SQL Server that guarantees all secondary nodes are up to date with the primary node. Synchronous commit mode can be useful when data loss is unacceptable. Every time a primary database receives a write, each of the secondary databases quickly replicate the same action, so as to keep up to date with the primary. The downfall of Synchronous commit mode is that a database administrator cannot quickly recover from a secondary database if the primary database fails due to a bad write or delete. Instead the database administrator only has multiple copies of the same corrupt database. Synchronous Commit Mode also suffers from a performance penalty, because the primary needs to wait for Secondary's to become synchronized before carrying out too many operations. Database administrators should also keep in mind that synchronization is not always guaranteed.

There are many factors that can affect Synchronization. According to Microsoft Documentation these factors include,

“The synchronized secondary replica will remain healthy unless one of the following occurs:

A network or computer delay or glitch causes the session between the secondary replica and primary replica to timeout.

You suspend a secondary database on the secondary replica [...]

You add a primary database the availability group [...]

You change the primary replica or the secondary replica to asynchronous-commit availability mode [...]

You change any secondary replica to synchronous-commit availability mode [...]

If any of the above happen the performance penalty incurred by the database could be for naught due and loss of data could still occur. When working with important data, it is necessary to quickly respond to any of the above.

Although there could be a failure to maintain synchronization, the benefits of synchronous commit mode can still be extremely useful. Having exact copies of the primary node could be useful in situations like server outages or network failures. In situations where great data loss could occur, an up to date secondary node could experience zero data loss. Synchronous commit mode also has more failover options than asynchronous commit mode. Synchronous commit mode can take advantage of, manual failover, and automatic failover.

## Asynchronous Commit Mode

Asynchronous commit mode is an availability mode that allows the secondary nodes to have a delay after changes are written to the primary. This delay means that there is an interval of time between an action being carried out on the primary node, and the secondary node. Asynchronous commit mode can prevent certain actions from disrupting the database, and all of its secondary nodes. Because asynchronous commit mode has a delay between the primary and second nodes, data can be lost in the event of a failover. It is possible that when a primary node fails not all of the changes have been reflected in each of the secondary nodes. As a result, the secondary node that takes over as the primary node may not have all of the changes as the previous primary node. This data loss is an important thing to keep in mind when choosing a commit mode for an SQL Server database. It is usually better to incur some data loss than total data loss, however, so choosing an Asynchronous commit mode is better than having no back up at all. If the data is not critical, Asynchronous commit mode may be the best option available as it reduces the latency inherent in Synchronous-commit mode. It is also important to note that Asynchronous commit mode only supports forced failover.

## Automatic Failover

When Synchronous-Commit Mode is configured with automatic failover, downtime of a database after the loss of a primary node is minimal. A replica set is specified as a failover replica, and when a primary node is lost, the secondary node specified as the failover replica quickly takes over as the primary. In a failover situation, there is no need for a database administrator to intervene in the failover process giving it an advantage over other failover options. Another benefit of using automatic failover with synchronous-commit Mode is that the database can maintain a near zero data loss during the failover process. Choosing this combination allows for optimal availability and minimal data loss. [2]

## Manual Failover

Manual failover occurs in a similar manner to automatic failover, but with the database administrator's intervention. Before a failover occurs, there must be a secondary replica that is synchronized with the primary node via synchronous commit mode. Once a primary becomes compromised in some way, the database administrator can manually failover to a secondary node. When a primary is signaled by the WSFC cluster, it will go offline, and the secondary node will finish committing changes from the log thus resulting in nearly zero data loss. If any uncommitted changes are still present

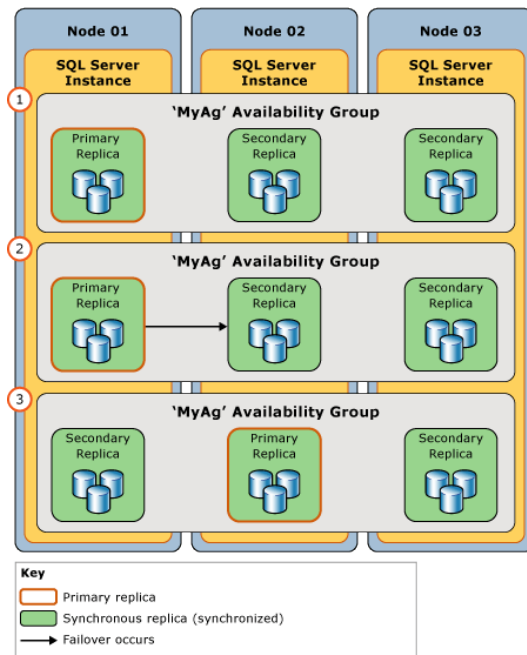


Figure 2. Manual Failover

in the new primary node they will be rolled back automatically. Figure two shows the manual failover process. [2]

In the first row of figure 2, the primary replica is set up and currently handling all write and read requests. The secondary replicas are synchronized with the primary replica at all times. In row two the primary replica fails at which point the database administrator begins the manual failover process, and fails over to one of the secondary replicas. This secondary replica will then become the new primary after carrying out the

rest of the transaction log. If at this point there are still uncommitted changes, the new primary replica will roll them back which may cause other secondary's to be put in an unsynchronized state, until it can be synchronized again. If the original primary replica recovers, it can become a new secondary replica.

### Forced Failover

Forced Failover is generally a last resort in SQL Server. It should be used when a planned manual failover is not possible. Forced failover is not ideal, because the database could experience severe data loss, but is necessary when the WSFC cluster state is lost and manual failover is not an option. This loss of the WSFC cluster creates a need for failover as the primary node within that cluster could be compromised. According to Microsoft's documentation, you can avoid data loss by performing the fail over on a node in synchronized commit mode, as the data will be mostly up to date. There are situations, however when a forced failover is not possible. A forced failover requires a WSFC cluster to have a quorum, and the database administrator must be able to access the secondary server that they plan on failing over to. Microsoft encourages only using forced failover after the primary is no longer running, as clients could still remain connected to the old primary when the failover switches to the new primary. Figure 3 is an illustration that shows what happens when a forced failover is triggered, and fails over to a remote data center. [2]

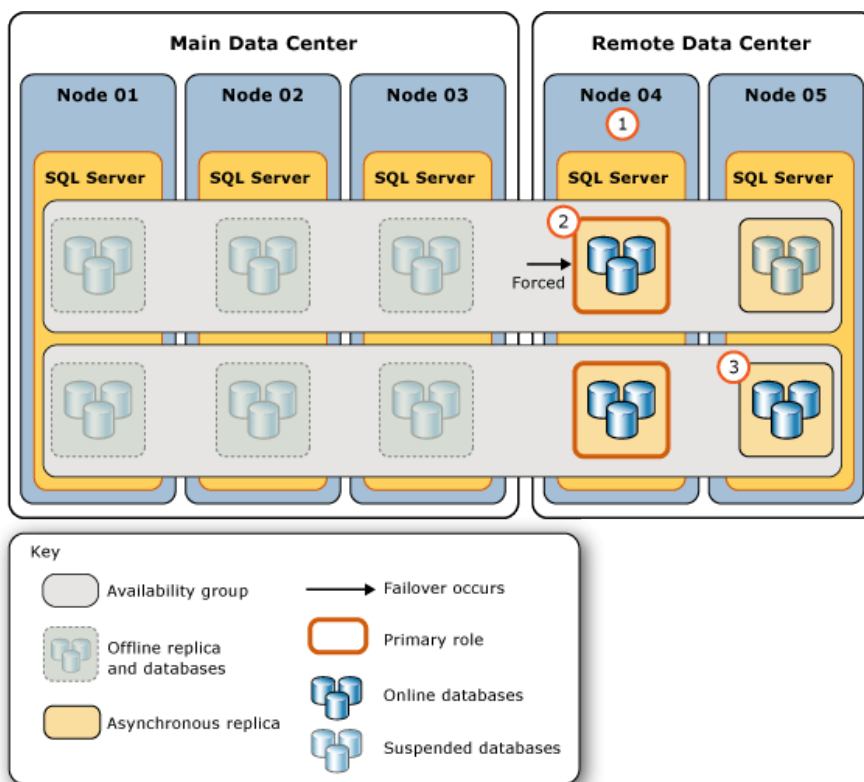


Figure 3. Forced Failover

In figure 3 the Main Data Center was the primary node for the cluster, and one of the secondary nodes in the remote data center is the node that the database



administrator is failing over to. In this case the primary node in the Main data center was compromised, so the database administrator made sure that the primary lost connection to insure no clients were connected to it before failing over to the remote datacenter. This failover method can be extremely useful in situations where a primary node fails un-expectantly, though it is not the safest of the failover methods, and can incur data loss.

### Benefits of Failover Clustering in SQL Server

SQL Server provides a lot of built in functionality in the case of a failover. If availability is the top priority in a database decision, then SQL Server may be a viable option. It is important to note, however, that SQL Server has absolutely no option to cluster for performance. If later on, the database becomes sluggish, there are no options for a performance enhancing cluster. If performance is not a primary concern, will not be a concern at any point in the future, and the database must maintain extremely high availability, then SQL Server provides all of the features necessary. If the primary purpose of the database, however is to scale horizontally, and increase performance as the data set grows, it may be worthwhile to consider MongoDB, which includes built in ways to both scale and increase performance at the same time.

## Clustering in MongoDB

MongoDB is a No-SQL Document database. Although No-SQL and SQL databases are vastly different, many of the features of each database are quite similar. For example, MongoDB provides a way to cluster for both performance and availability. Replication, or clustering for availability in MongoDB, is quite similar to clustering for availability in SQL Server. MongoDB provides two major clustering methods sharding (clustering for performance), and replication (clustering for availability). When used properly, both of these clustering techniques can vastly improve the overall functioning of the database. Both of these clustering options require careful fine tuning before implementation and can still fail even if configured correctly the upfront. Clustering in MongoDB also requires significant maintenance and close monitoring otherwise the database could slow to a halt and all performance or availability gained will be lost and could even end up drastically harming the database.

### Clustering for Performance (Sharding)

Sharding is MongoDB's method of clustering for performance. Sharding can be an difficult to do correctly, but can increase performance significantly. Sharding consists of three major components, shards, a mongos, and a config server. A shard is a subset of the total data in the database, the mongos is the query router that directs queries between shards, and the config server has important metadata for the database. All of these aspects work in conjunction to speed up queries, and balance the query load across shards evenly. Sharding, however does not guarantee better performance. Shards should be as even as possible and if they are not, queries will take longer and longer to process over time. Sharded clusters must also be monitored closely to ensure that data is evenly spread across all shards. One of the benefits of sharding is its ability to scale horizontally. If the database grows beyond the capacity of a server, another shard can be added and chunks can be automatically migrated across all shards.

There are a number of options that must be configured in order to have a performant MongoDB sharded cluster. The first is a Shard key. The way sharding works, is shard keys are assigned at the Collection level. A collection in this context can be considered similar to a table in a relational database. Each collection has a Shard key value. This value is extremely important and will be used to evenly distribute data amongst all shards. The mongos will use the shard key to route traffic for both reads and writes. Choosing a shard key can be a complicated process. All collections in a Sharded cluster must have a shard key. There are two Shard key types. The first is a hashed shard key and the second is a ranged shard key. A hashed shard key uses a hashing function to distribute the shard key values into chunks evenly. A ranged shard key works by grabbing a range of shard keys and putting them into a single chunk. Once the shard key type is chosen, the Mongos will be able to quickly route traffic to the necessary chunks.

Chunks are sections of distributed data. A chunk is a group of collections, and is where the mongos will route queries. Database administrators are able to choose a custom chunk size. The smallest chunk possible is a chunk containing only one shard key. Chunks of the smallest possible size cannot be split. Chunk size is another extremely important choice when setting up a sharded cluster, and can directly affect

performance. If a small chunk is chosen, the distribution of data will be much higher, but will cause more migrations than a large chunk size. A large chunk results in less frequent migrations, and is more efficient in terms of database overhead for the mongos. It is easier for a mongos to route to a larger chunk, because it contains more shard key values. For example, if a chunk is very small, and that chunk starts to exceed its given size, then the mongos needs to “migrate” the chunk, meaning it needs to redistribute the data in that chunk in order to keep the total size of the chunk small. When a migration is necessary the mongos initiates a balancing round. Any mongos can initiate a balancing round. The smaller the chunk size, the more frequently the mongos will have to initiate a balancing round.

When a balancing round is initiated, the cluster balancer starts relocating chunks in a collection in order to make the chunks even. The cluster balancer activates when it exceeds the migration threshold which is configurable by the database administrator. This process is time consuming, and can affect overall performance. The frequency of migrations is directly correlated to chunk size. Chunks larger than a single shard key can be split. Automatic splitting is on by default. All chunks have a maximum size, so when a chunk exceeds their maximum size the chunk is split by shard key. Because chunks are automatically split by shard key, a chunk of the smallest size, a single shard key, cannot be split. If data grows inconsistently and a chunk is split all the way to the smallest size, but continues to grow, then the chunk becomes what is known as a Jumbo Chunk. This can be a major issue, because no migration can help redistribute the data, and the entire sharded cluster needs to be re-sharded and redistributed in order to alleviate the imbalance.

There are two major shard key methods in MongoDB. The default sharding method is Ranged sharding. Ranged sharding is ideal for when the shard key values are evenly distributed by default. When the shard keys tend to grow closer together it may be much better to implement Hashed sharding. The decision of which sharding method to use, is extremely important in maintaining a fast and performant cluster, and could vary from database to database.

Hashed Sharding is a shard key method that uses a hashed index for the shard key. In theory hashed sharding creates a more even distribution of chunks. The downfall of hashed sharding is reduced Query Isolation. Query Isolation is the mongos’ ability to resolve a query to a single shard. A reduced query isolation means a longer response time, because the mongos has to query all shards. Hashed sharding means the mongos is more likely to use a broadcast operation. Broadcast Operations are slower than targeted operations, because when the mongos needs to resolve a query it queries all shards and merges the results before it is returned. These broadcast operations can cause drastically more overhead than a targeted operation, because in a targeted operation the mongos knows exactly which shard the query needs to route to. According to the MongoDB documentation it is preferable to use targeted operations over broadcast operations. Hashed Sharding is good to use when the shard key is “monotonically increasing.” A monotonically increasing key is one that gradually increases, until the keys start to merge on each other. Thanks to the hash function performed on the key, hashed sharding values that begin to converge are often no

longer close in value post-hash.

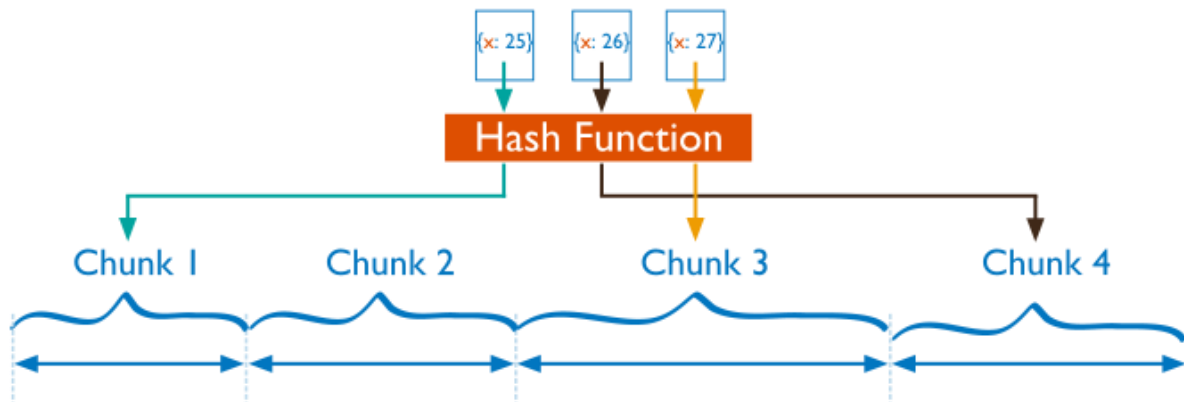


Figure 4. Hashed Sharding

Figure 4 shows an illustration of what hashed sharding looks like. The x values represent the shard key used for each query. The x value is then hashed and directed to the correct chunk. It is apparent from the illustration that the x values, even though they are close in value, are distributed in completely different chunks. It is situations like the above, where the shard keys are close together, that hashed sharding is much more effective than ranged hashing.

Ranged Sharding on the other hand is ideal for shard keys that are already evenly distributed. With ranged sharding, values are closer in a chunk if the shards are close, unlike hashed sharding, which loses proximity due to the hash function. With values that are closer in value it is easier for the balancer to find shards that are close together. When the mongos knows the values are close by it can query much more efficiently than if there is no direct pattern. According to the MongoDB documentation,

“Ranged sharding is most efficient when the shard key displays the following traits:

- Large Shard Key Cardinality
- Low Shard Key Frequency
- Non-Monotonically Changing Shard Keys“ [3]

So even though ranged sharding is on by default it is important to ensure that the shard keys chosen have all of the above characteristics, otherwise the values could monotonically increase, because there is no hashing function to keep all of the shard keys distributed evenly.

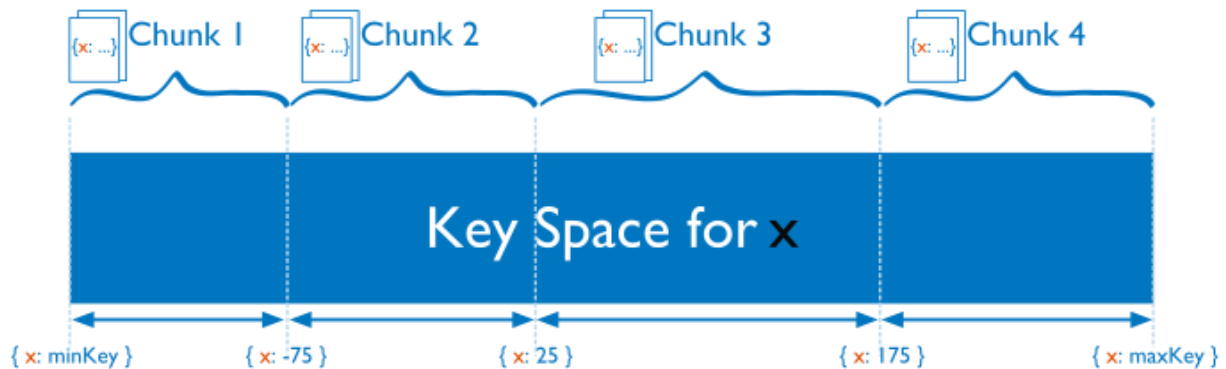


Figure 5. Ranged Sharding

Figure 5 shows an example of how ranged sharding works. The key,  $x$ , is distributed to a chunk based on its value. If  $x$  is between `minKey` and `-75`, then  $x$  goes to chunk one. As  $x$  grows, it is placed into higher and higher chunks. This is convenient, and fast if the shard key values are well distributed on their own. Queries on ranged shards are generally more efficient than a database sharded with a hashed shard, because the mongos can more efficiently query the database using, targeted operations. When possible, the database creator should prefer ranged keys with well distributed shard keys, because the mongos will not have to use broadcast queries nearly as much as it would with hashed sharding.

Ultimately when creating a Mongo database it is important to make the correct decisions up front even though it is fairly easy to re-shard, and modify most of the options later on down the line. It is also important to closely observe and maintain the sharded cluster; some companies even have alert systems in place just in case shards grow unevenly, because if not corrected quickly, a bad shard could stop an entire database and even take down an entire server.

### Clustering for availability (Replication)

MongoDB provides a built in way to cluster for availability called replication. Replication in MongoDB is actually very similar to the way SQL Server handles clustering for availability. Each replicated cluster in MongoDB has a primary node with multiple secondary nodes. The secondary nodes each read from the rolling oplog and then replicate the actions recorded in order to keep themselves up to date with the primary node. It is important to keep in mind that all reads and writes are directed to the primary, unless otherwise specified. Each secondary node is what is called a replica set. Replica sets are identical to the primary node. Every time a primary node receives a write or update, the same write or update is written to the oplog. The oplog is then stored in the cache of the secondary nodes, and written to the replica set, thus keeping the secondary node up to date.

MongoDB also allows reads to be performed from secondary nodes though it is strongly advised against as it is highly likely that queries will grab “stale data”, or data that has been changed due to the lag between the oplog being written and the oplog

being replicated on the secondary node. There are a few edge cases where it may make sense to perform reads on secondary nodes, like reducing lag with geographically distant replica sets, but in most cases if performance is the goal, it does not make sense to read from secondary nodes because the queries will most likely be bottle necked by the primary node anyway. If performance is critical it is recommended to use a sharded cluster instead, because sharded clusters provide a safer and more efficient way to cluster for performance.

Every Replication cluster must have a primary node, and one or more secondary nodes. Each node, as discussed, contains a direct copy of the primary node. All nodes in the cluster send what are referred to as, heartbeats. A heartbeat is sent every second, and a response must be received within ten seconds. If a heartbeat is not responded to within ten seconds, the node will be considered dead.

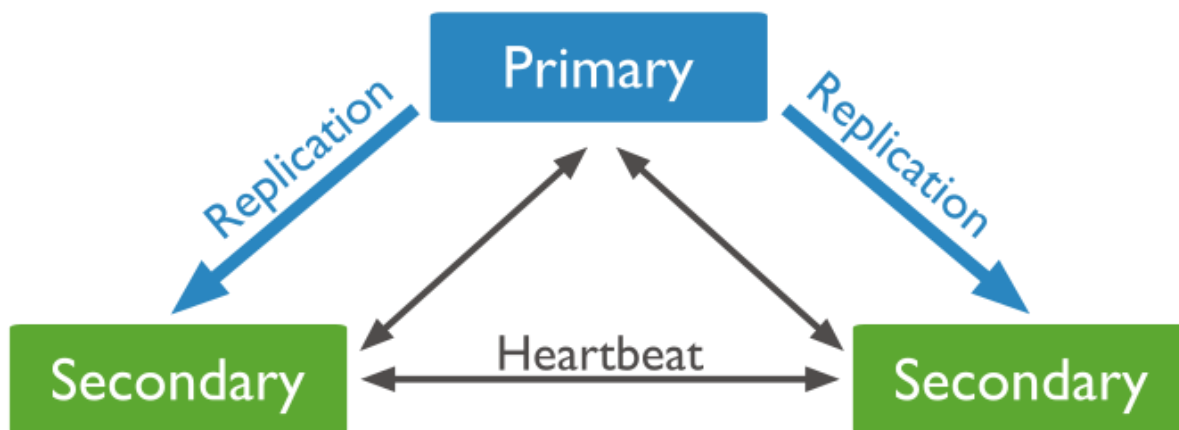


Figure 6. Replication Diagram

If a primary node loses connection to the majority of its primary nodes, then a new primary node must be chosen via an election. An election is a vote between up to 7 of the secondary nodes to choose a new primary.

When a primary node is lost an election is necessary to figure out which node is most qualified to be the new primary node. There are many factors that contribute to the election. In an election, up to seven nodes can have a single vote. The winner of the election is based on a number of factors. Even though only seven nodes can vote, each cluster can have more than seven nodes. This means that if there are more than 7 nodes only seven can vote, and the remaining nodes simply do not get a vote. In the case that there are an even set of nodes, which leaves a greater possibility for a tie, an arbiter can be created. An arbiter is simply a node that exists to break ties. The reason arbiters exist is because they cause less overhead than an entire replica set. Voting factors are based on the following:

Heatbeats – if a heartbeat is lost the remaining voters will not vote for the node whose heartbeat will not return.

Priority Comparisons – If a nodes priority is higher than other nodes it is more likely to be voted the new primary.

**Optime** – The optime is the time of the latest oplog operation applied to the node. The node with the most recent optime is more likely to be voted for than a node with an older optime.

**Connections** – If a node cannot connect to the majority of the nodes in a cluster votes will not be cast for that node.

**Loss of a Data Center** – If an entire data center is compromised it is unlikely the remaining nodes at that data center will be available to be a primary.

**Network Partition** – Sometimes a node is separated into a network partition. A network partition is a small subset of nodes. Sometimes a primary node will still think it is a primary node when put into a network partition, but since, by definition, a network partition is a small subset of the total nodes in the cluster, that primary node is no longer eligible to be voted the primary node of the cluster.

There are many things that can cause an election to take place besides the loss of a primary node. For example, a vote can be triggered if a new replica set is created or a primary steps down to secondary status. The following events can cause a primary node to step down to secondary status:

- The primary node receives a `replSetStepDown` command
- A secondary node is eligible and has a higher priority than the current primary node
- The primary realizes that it is not currently connected to the majority of secondary nodes anymore [3]

There are further stipulations in elections. Each node, even non-voters, can veto an election, according to MongoDB documentation, for the following reasons:

“If the member seeking an election is not a member of the voter’s set.

If the member seeking an election is not up-to-date with the most recent operation accessible in the replica set.

If the member seeking an election has a lower priority than another member in the set that is also eligible for election.

If a priority 0 member is the most current member at the time of the election. In this case, another eligible member of the set will catch up to the state of this secondary member and then attempt to become primary.

If the current primary has more recent operations (i.e. a higher optime) than the member seeking election, from the perspective of the voting member.

If the current primary has the same or more recent operations (i.e. a higher or equal optime) than the member seeking election.” [3]

All of these voting stipulations are meant to insure that the best node is elected to the primary node every single time. If the vote goes well, the best node will be chosen as the new primary node, and hopefully another election will not need to happen for some time.

Maintaining a performant cluster, whether it be a sharded cluster or replication cluster can be a complicated process, but when done correctly can provide massive performance or availability gains. Although the cluster can be incredibly more robust, they require precise setup and maintenance. A MongoDB cluster should not be setup and then left to run on its own for long periods of time, and thus, depending on the needs of the database, and the manpower available to the database, may not be the best option in all situations.

### Foursquare

Foursquare is an app that used to allow users to check in to locations that they frequented (e.g. restaurants, the gym, stores). Foursquare was created in 2008, and used MongoDB as their primary database. Foursquare was able to take advantage of MongoDB's horizontal scaling via sharded clusters, and allowed them scale to over sixty million users [4]. MongoDB was ideal for their product, because they were handling millions upon millions of check-ins, and user activities a day. Mongo allowed them to maintain quick performance while also scaling horizontally. Although MongoDB worked out for the most part, just two years after their launch in 2010 Foursquare experienced seventeen hours of downtime due to poor database design related to sharded clusters.

The Foursquare database outage is an example of how difficult it is to create an evenly distributed database using a sharded cluster, and proves how careful one needs to be when creating the shards, shard keys, chunk size etc. In the "post-mortem" shared publicly on Google+, they revealed the issue had to do with unchecked shards over time. Data was not growing consistently among all shards, and over time one shard in particular became much larger than the others. The disproportionately large shard caused server issues when memory quickly got out of hand,

"On Monday morning, the data on one shard (we'll call it shard0) finally grew to about 67GB, surpassing the 66GB of RAM on the hosting machine. Whenever data size grows beyond physical RAM, it becomes necessary to read and write to disk, which is orders of magnitude slower than reading and writing RAM. Thus, certain queries started to become very slow, and this caused a backlog that brought the site down.

We first attempted to fix the problem by adding a third shard. We brought the third shard up and started migrating chunks. Queries were now being distributed to all three shards, but shard0 continued to hit disk very heavily. When this failed to correct itself, we ultimately discovered that the problem was due to data fragmentation on shard0. In essence, although we had moved 5% of the data from shard0 to the new third shard, the data files, in their fragmented state, still



needed the same amount of RAM. This can be explained by the fact that Foursquare check-in documents are small (around 300 bytes each), so many of them can fit on a 4KB page. Removing 5% of these just made each page a little more sparse, rather than removing pages altogether” [1]

Eventually the Foursquare engineering team realized they would need to add another shard, and re-distribute the data, and re-shard their entire database. Essentially, what happened in the case of Foursquare is, their sharded database seemed to be distributed pretty evenly across shards. Over time the data growth within the sharded cluster grew unevenly resulting in one shard being far more massive than the rest, eventually filling up the RAM on their server, and thus resorting to the disk, a far slower method of retrieving data, for all of its reads and writes. When on a scale as massive as Foursquare, a sharding mistake can cost millions.

Ultimately this incident could have been prevented by keeping a careful watch on the database, as the data set grew. The solution Foursquare came up with was to write an alert program that notified them if the database started growing asymmetrically, add a shard, and re-shard the entire database. Foursquare employee Eliot mentioned some points about how this situation could have been avoided,

“The main thing to remember here is that once you’re at max capacity, it’s difficult to add more capacity without some downtime when objects are small. However, if caught in advance, adding more shards on a live system can be done with no downtime.”

For example, if we had notifications in place to alert us 12 hours earlier that we needed more capacity, we could have added a third shard, migrated data, and then compacted the slaves.” [1]

Obviously this situation could have been prevented had they taken those steps earlier, but Eliot also mentions that this instance was more of an edge case than expected behavior, because part of the issue was having documents that were “too small” and caused issues when they could not be dropped from memory,

“Most sharded deployments will not meet these criteria. Anyone whose documents are larger than 4KB will not suffer significant fragmentation because the pages that aren’t being used won’t be cached.” [1]

So even though specific edge cases were encountered to crash Foursquare’s database it shows how important it is to not only have a consistent shard key distribution, but to make sure it grows in an equally distributed manner.

### Benefits of Clustering in MongoDB

Although SQL Server and MongoDB are in two vastly different classifications of databases, the main reason a user would want to choose MongoDB in regards to clustering is for the option of sharding. In many ways MongoDB and SQL Server handle

high availability clustering in the same way, although SQL Server being an SQL database by nature, and more time-proven than MongoDB may be the more obvious option based on consistency alone. If the main concern for the database is performance, and low latency while scaling then MongoDB makes more sense before taking all of the pros and cons of No-SQL and SQL databases into account.

### Conclusion/Recommendation

The choice of database should be based on far more than clustering alone, though it could be an important factor in the decision making process. Clustering of one database may provide a massive advantage technologically, and be more convenient for a development team, and could make things like scaling much more efficient in the long run.

MongoDB may be the obvious choice if horizontal scaling, and performance on large data sets is necessary. The combination of built-in sharding, automatic migration, and the ability to quickly add servers when necessary with minimal downtime would be a huge advantage. If replication is necessary MongoDB has built in methods of dealing with automatic failovers via replicated data sets. It may be more convenient in the future to deal with a quickly growing data set in MongoDB, but sharded clusters do still require monitoring and maintenance as shown in the Foursquare database outage.

SQL Server is useful for when you need a database that offers consistency and availability. SQL Server has high availability cluster options, and multiple different ways of dealing with failover. The downside of SQL Server is that it has no built in methods of increasing performance via clustering.

When choosing a database, it is important to choose the database for the job. Sometimes it may not matter which database is chosen, but often times it comes down to features included. If the features specific to MongoDB or SQL Server meet the needs of the project, then either one will operate well.

## References

[1] *Google Groups*. (2016). *Groups.google.com*. Retrieved 7 December 2016, from <https://groups.google.com/forum/#!topic/mongodb-user/UoqU8ofp134>

[2] *Overview of Always On Availability Groups (SQL Server)*. (2016). *Msdn.microsoft.com*. Retrieved 7 December 2016, from <https://msdn.microsoft.com/en-us/library/ff877884.aspx>

[3] *Sharding — MongoDB Manual 3.2*. (2016). *Docs.mongodb.com*. Retrieved 7 December 2016, from <https://docs.mongodb.com/v3.2/sharding/>

[4] Weber, H. & Novet, J. (2016). *Foursquare by the numbers: 60M registered users, 50M MAUs, and 75M tips to date*. *VentureBeat*. Retrieved 7 December 2016, from <http://venturebeat.com/2015/08/18/foursquare-by-the-numbers-60m-registered-users-50m-maus-and-75m-tips-to-date/>

## Figures

Figure 1. Windows Failover Clustering Illustration from msdn documentation [2]

Figure 2. Manual Failover illustration from msdn documentation [2]

Figure 3. Forced Failover illustration from msdn documentation [2]

Figure 4. Hashed Sharding illustration from MongoDB Documentation [3]

Figure 5. Ranged Sharding illustration from MongoDB Documentation [3]

Figure 6. Replication illustration from MongoDB Documentation [3]