ALTOROS™

MongoDB vs. Couchbase Server:

Architectural Differences and Their Impact

This 45-page paper compares two popular NoSQL offerings, diving into their architecture, clustering, replication, and caching.

By Vladimir Starostenkov,
R&D Engineer at Altoros

Q3 2015

Table of Contents

1. OVERVIEW 3
2. INTRODUCTION 3
3. ARCHITECTURE 4
4. CLUSTERING 4
4.1 MongoDB 4
4.2 Couchbase Server 5
4.3 Partitioning 6
4.4 MongoDB 7
4.5 Couchbase Server 7
4.6 Summary 7
5. REPLICATION 8
5.1 MongoDB 8
5.2 Couchbase Server 10
5.3 Summary 12
6. CACHING 13
6.1 MongoDB 13
6.2 Couchbase Server 13
6.3 Impact on Performance 13
7. CONCLUSION 14
8. APPENDIX A. NOTES ON MAIN TOPICS 14
8.1 Notes on Topology 14
8.2 Notes on Partitioning 15
8.3 Notes on Couchbase Replication 21
8.4 Notes on Availability 21
8.5 Notes on Read Path 25
8.6 Notes on Durability 28
9. APPENDIX B. NOTES ON MEMORY 32
9.1 MongoDB 32
9.2 Couchbase Server 34
10. APPENDIX C. NOTES ON STORAGE ENGINE 38
10.1 MongoDB 38
10.2 Couchbase Server 41
11. ABOUT THE AUTHORS 45

1. Overview

Organizations integrating distributed computing into their operations face the challenge of adopting a database that is up to the new challenges presented by a new architecture. Whether virtualizing local resources, scaling horizontally, or both, a distributed computing architecture requires new thinking about how to handle flows of un-structured and semi-structured data, often in real-time.

The development of open-source NoSQL database options addresses this challenge. NoSQL—generally meant to stand for “Not only SQL”—provides a method to handle these new data flows reliably and at scale.

Two of the most prominent NoSQL offerings come from MongoDB and Couchbase. Both are popular, both perform well, and both have major customers and strong communities.

2. Introduction

MongoDB and Couchbase Server are the two most popular document-oriented databases, designed to handle the semi-structured increasingly found in enterprise cloud computing and Internet of Things deployments. They belong to a new generation of NoSQL databases.

They are also both considered to be CP systems within the well-known CAP theorem system postulated by Professor Eric Brewer of UC-Berkeley. However, they both have options to behave like an AP system.

The letters in the CAP acronym represent:

- Consistency (the latest information is always available everywhere)
- Availability (every read and write request receives a response)
- Partitioning Tolerance (which can be thought of as a form of fault tolerance)

The CAP theorem states that a database cannot simultaneously provide all three of the above guarantees. Practically speaking, NoSQL databases are forced to choose whether to favor consistency or availability in specific scenarios.

Within this context, CP systems, such as MongoDB and Couchbase Server, can be adjusted to favor availability via settings and configuration options that change them to be AP systems such that data is *eventually consistent*, often within milliseconds. MongoDB and Couchbase Server rely on asynchronous persistence for performance but support synchronized persistence when necessary. They also have options to weaken consistency for availability.

This paper identifies key architectural differences between the two databases. It then highlights the impact of these architectural decision on the performance, scalability, and availability of each.

3. Architecture

This white paper is not a performance benchmark report. Instead, it offers an architect's view as well as a deep technical comparison of how the two databases go about their tasks.

Some conclusions are offered, with the caveat that selection of a NoSQL database for any individual enterprise or project must be based on the unique goals, needs, and expectations faced by executives, project managers, technical evaluators and decision makers, product owners, etc.

This white paper was written by a joint team of Altoros engineers and technical writers from the company's Sunnyvale, CA headquarters and Minsk Development Center. It starts with an examination of clustering, which forms the key difference between MongoDB and Couchbase Server. It then looks at the implications of clustering on availability, scalability, performance, and durability.

Additional technical information on these topics is provided in Appendix A. Supplemental information on memory and storage is provided in Appendix B and C.

4. Clustering

The most fundamental feature of a distributed system is its topology—the set of properties that do not change under system scaling. Whether there are 10 or 1,000 nodes in a cluster, the node types, their roles, and connection patterns stay the same.

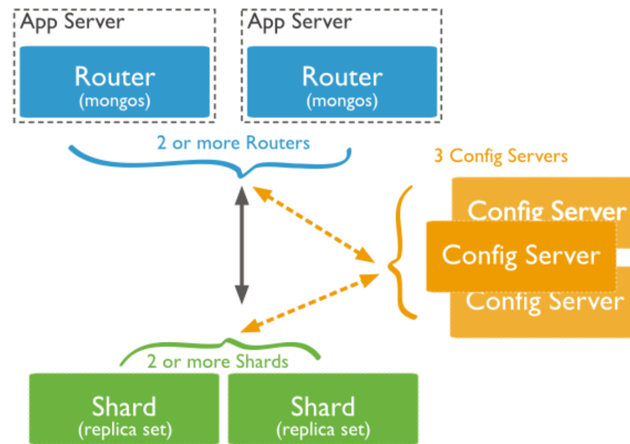
By looking at the topology, we can understand the system components and the information flows among them.

The two databases under examination here offer profoundly different topologies. MongoDB takes a hierarchical, master/slave approach, while Couchbase Server has a peer-to-peer (P2P) topology.

4.1 MongoDB

MongoDB deployment architecture is very flexible and configurable, with a dominating *hierarchical* topology type. Final deployment properties with respect to system performance and availability depend on the database administrator experience, various community best practices, and the final cluster purpose.

We will consider a sharded cluster here, suggested in the official MongoDB documentation as the deployment best capable of scaling out.



A sharded cluster has the following components: shards, query routers, and config servers. Each component is installed on a separate server in a production environment.

Shards store the data. To provide high availability and data consistency each shard is a replica set—a number of mongod processes located on different physical machines. The replica set is itself a hierarchical structure having primary and secondary nodes within it.

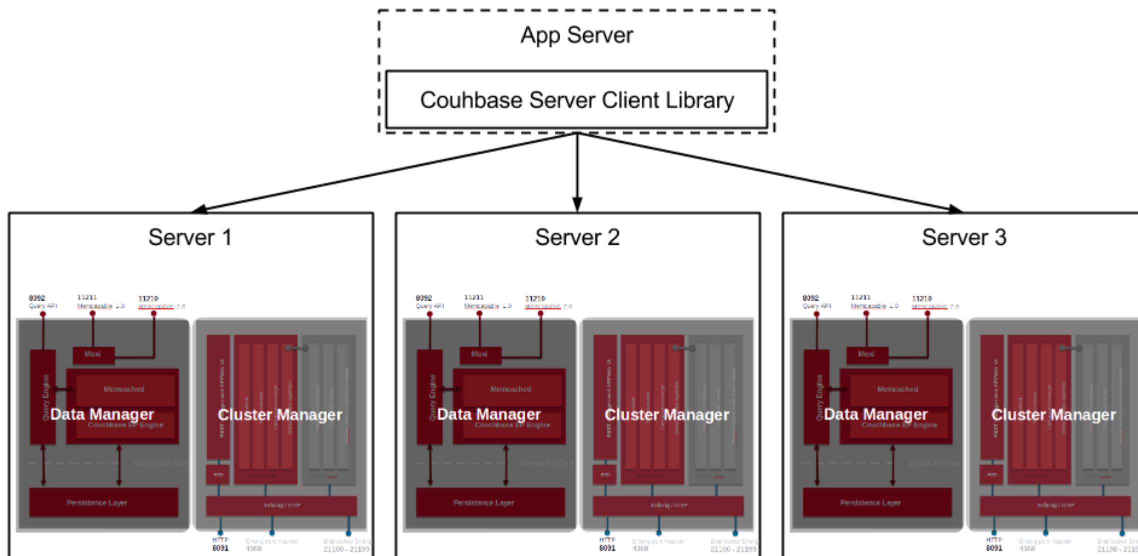
Query Routers, or mongos instances, interface with client applications and direct operations to the appropriate shard or shards. They process queries, target operations to appropriate shards and then return results to the clients. A sharded cluster usually contains more than one query router to divide the client request load. A client sends requests to one query router.

Config servers store the cluster's metadata. This data contains a mapping of the cluster's data set to the shards. The query router uses this metadata to target operations to specific shards. Production sharded clusters have exactly three config servers.

4.2 Couchbase Server

Couchbase Server has a flat topology with a single node type. All the nodes play the same role in a cluster, that is, all the nodes are equal and communicate to each other on demand.

One node is configured with several parameters as a single-node cluster. The other nodes join the cluster and pull its configuration. After that, the cluster is operated by connecting to any of the nodes via Web UI or CLI.



Going deeper into the details, we see each node runs several processes logically grouped into a Data Manager and Cluster Manager. The Data Manager serves client requests and stores data. The Cluster Manager is responsible for node and cluster-wide services management. It monitors each node in the cluster, detects node failures, performs node failovers and rebalance operations, etc.

There are additional brief notes on this topic in Appendix A.

4.3 Partitioning

NoSQL databases were created to handle large data sets and high throughput applications that challenge the capacity of a single server. Users can scale up vertically by adding more resources to a server, scale out horizontally by adding more servers, or both.

Horizontal scaling is proving popular and treats servers as a commodity. It divides the data set and distributes the data over multiple servers, with all the servers collectively making up a single logical database.

The *partitioning* of data thus emerges as a key issue. Both Couchbase Server and MongoDB were designed with commodity hardware in mind and strongly rely on data partitioning. The two database programs take different approaches in treating the data logically (viewed as in a single system), physically, and process-wise.

This information, along with additional extensive notes, is found in Appendix A.

4.4 MongoDB

MongoDB partitions data sets into chunks with a chunk storing all data within a specific range, and assigns chunks to shards. By default, it creates two 64 megabyte chunks per shard. When a chunk is full, MongoDB splits it into two smaller 32 megabyte chunks. To ensure an even distribution of data, MongoDB will automatically migrate chunks between shards so that each shard has the same number of chunks.

This is why sharded clusters require routers and config servers. The config servers store cluster metadata including the mapping between chunks and shards. The routers are responsible for migrating chunks and updating the config servers. All read and write requests are sent to routers because only they can determine where the data resides.

4.5 Couchbase Server

Couchbase Server partitions data into 1,024 virtual buckets, or vBuckets, and assigns them to nodes. Like MongoDB chunks, vBuckets store all data within a specific range. However, Couchbase Server assigns all 1,024 virtual buckets when it is started, and it will not reassign vBuckets unless an administrator initiates the rebalancing process.

Couchbase Server clients maintain a cluster map that maps vBuckets to nodes. As a result, there is no need for routers or config servers. Clients communicate directly with nodes.

4.6 Summary

Impact on Availability

Couchbase Server applications read and write directly to database nodes. However, MongoDB applications read and write through a router. If the router becomes unavailable, the application can lose access to data.

Couchbase Server topology configuration is shared by all nodes. However, MongoDB topology configuration is maintained by config servers. If one or more config servers become unavailable, the topology becomes static. MongoDB will not automatically rebalance data, nor will you be able to add or remove nodes.

Impact on Scalability

Couchbase Server is scaled by adding one or more nodes on demand. As such, it is easy to add a single node to increase capacity when necessary. However, MongoDB is scaled by adding one or

more shards. Therefore, when it is necessary to increase capacity, administrators must add multiple nodes to create a replica set.

Impact on Performance

MongoDB recommends running routers next to application servers. However, as the number of application servers increases, the number of routers increases and that can be a problem since the routers maintain open connections to MongoDB instances. Eventually, the routers will open too many connections and performance will degrade. The solution is to separate routers from application servers and reduce the number of them. While this alleviates the problem of too many open connections, it limits users' ability to scale routers by adding more.

While MongoDB routers can send only one request over a connection, Couchbase Server clients can send multiple requests over the same connection. Couchbase Server does not rely on the equivalent of routers, rather client send requests directly to nodes that own the data.

MongoDB will migrate only one chunk at a time in order minimize the performance impact on applications. However, migration happens automatically and will have some impact on performance. Couchbase Server will not migrate vBuckets unless an administrator initiates the rebalancing process. The enables administrators to plan when to rebalance data in order to avoid doing so during high workloads.

5. Replication

With multiple copies of data on different database servers, replication protects a database from the loss of a single server. Replication thus provides redundancy and increases data availability. It also allows recovery from hardware failures and service interruptions.

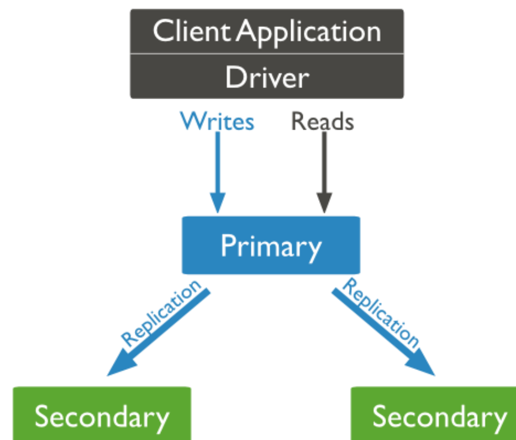
In some cases, replication is used to increase read capacity. Clients have the ability to send read and write operations to different servers. Maintaining copies in different data centers increases the locality and availability of data for distributed applications.

There are significant differences between how MongoDB and Couchbase Server handle this functionality, stemming from their respective topologies.

5.1 MongoDB

Increased data availability is achieved in MongoDB through *Replica Set,* which provides data redundancy across multiple servers. Replica Set is a group of mongod instances that host the same data set. One mongod, the primary, receives all data modification operations (insert / update / delete).

All other instances, secondaries, apply operations from the primary so that they have the same data set.



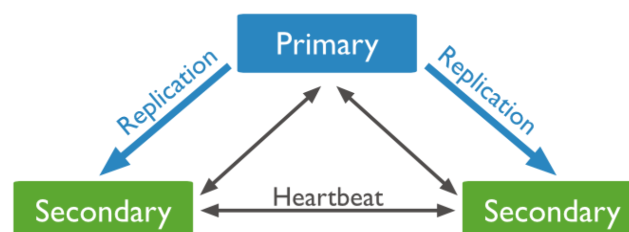
Because only one member can accept write operations, replica sets provide strict consistency for all reads from the primary. To support replication, the primary logs all data set changes in its oplog. The secondaries asynchronously replicate the primary's oplog and apply the operations to their data sets. Secondaries' data sets reflect the primary's data set.

It is possible for the client to set up Read Preference to read data from secondaries. This way the client can balance loads from master to replicas, improving throughput and decreasing latency. However, as a result secondaries may not return the most recent data to the clients. Due to the asynchronous nature of the replication process, replica read preference guarantees only eventual consistency for its data.

Failover

Within the replica set, members are interconnected with each other to exchange heartbeat messages. A crashed server with a missing heartbeat will be detected by other members and removed from the replica set membership. After the dead secondary recovers, it can rejoin the cluster by connecting to the primary, then catch up to the latest update.

If a crash occurs over a lengthy period of time, where the change log from the primary doesn't cover the whole crash period, then the recovered secondary needs to reload the whole data from the primary as if it was a brand new server.



In case of a primary DB crash, a leader election protocol will be run among the remaining members to nominate the new primary, based on many factors such as the node priority, node uptime, etc. After getting the majority vote, the new primary server will take its place.

Geo-replication

MongoDB relies on standard replication to support multi data center deployments. In fact, a multi data center deployment consists of a single cluster that spans multiple data centers. It does so by placing primaries and/or secondaries in different data centers. However, MongoDB is limited to unidirectional replication and does not support multi-master or active/active deployments. It can only replicate from a primary to one or more secondaries, and there can only be one primary per shard (or subset of data). While a subset of data can be read from multiple data centers, it can only be written to a single data center.

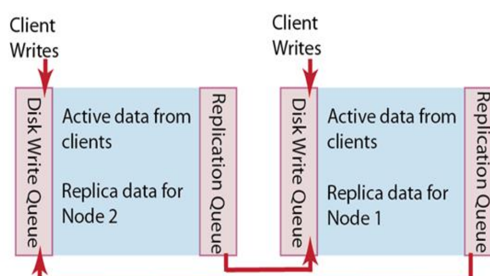
5.2 Couchbase Server

Couchbase Server nodes store both active and replica data. They execute read and write requests on the active data, the subset of data they own, while maintaining copies of data owned by other nodes (replica data) to increase availability and durability. Couchbase Server does not implement primary and secondary roles such that a node can store only active data or only replica data. When a client writes to a node in the cluster, Couchbase Server stores the data on that node, then distributes the data to one or more nodes within a cluster.

Replication within a single Couchbase Server cluster is configured through a single per-Bucket parameter—Replication Factor. As described earlier, Bucket is internally split into a number vBuckets; these vBuckets are evenly distributed across all the cluster nodes. Replica Factor is the number of copies of each vBucket that are evenly distributed across the cluster as well.

The cluster administrator can influence the vMap only by specifying the rack awareness configuration, which gives vBucket system an additional clue on how to distribute the active and replica vBuckets across the available cluster nodes.

The following diagram shows two different nodes in a Couchbase cluster and illustrates how two nodes can store replica data for one another:



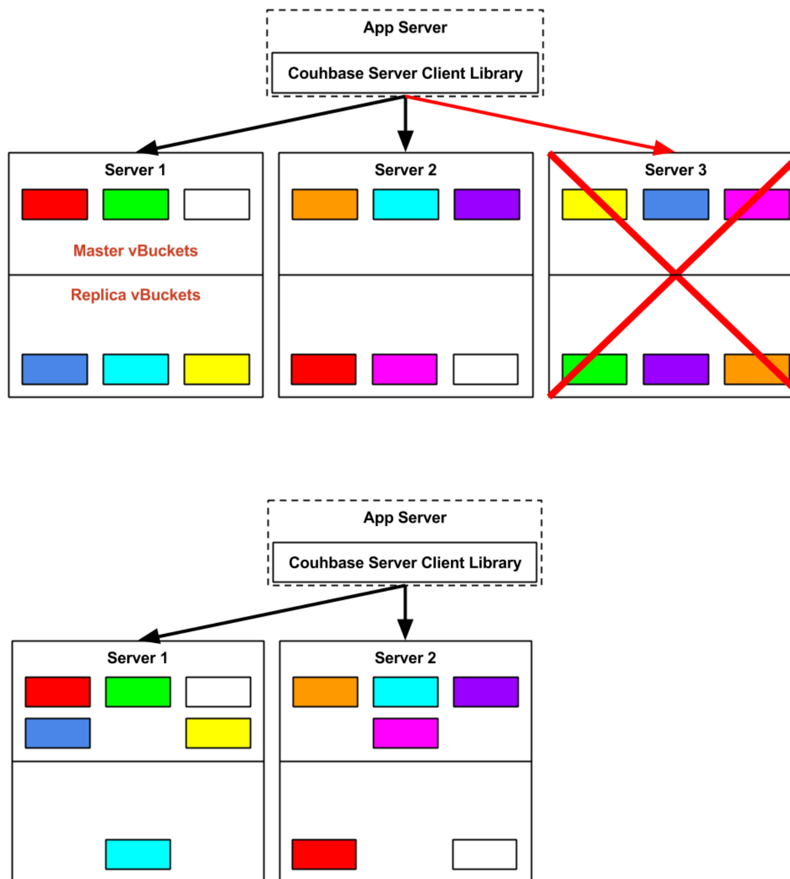
There are additional notes on Couchbase replication in Appendix A.

Failover

Failover is the process in which a node in a Couchbase Server cluster is declared as unavailable, and a replica vBucket is promoted to an active state to start serving write operations.

In case of node failure or cluster maintenance, the failover process contacts each server that was acting as a replica and updates the internal table that maps client requests for documents to an available Couchbase Server.

Failover can be performed manually or automatically using the built-in automatic failover process. Auto failover acts after a preset time, when a node in the cluster becomes unavailable.



On the picture above, one server goes down and three replica vBuckets are promoted for mastering the data after a failover process is finished. The load balance in this case is broken, as one server is mastering more data and gets more requests from clients. The rebalance process must be triggered in this scenario.

Geo-replication

Couchbase Server implements a specialized protocol for geo-replication. Unlike MongoDB, a multi data center deployment of Couchbase Server consists of independent clusters that replicate data to each other. It supports both unidirectional and bidirectional replication. In fact, any cluster can operate without any. As a result, it can be used not only for high availability and disaster recovery, but for full read and write locality via multi-master or active/active deployments—all data can be read or written from any data center.

5.3 Summary

Impact on availability

Couchbase Server supports multi-master, or active/active, multi-data center deployments to provide the highest form of availability. However, MongoDB does not. MongoDB can only execute writes for a specific subset of data in a single data center. With Couchbase Server, a failed data center would have no impact on the application as requests could be immediately sent to a different data center. However, with MongoDB, applications would have to wait until it promoted one of its secondaries to primary.

Impact on scalability

To support a new geographic location, administrators simply deploy a new cluster and replicate its data via XDCR. However, with MongoDB, the existing cluster has to be updated by adding a new shard and extending existing shards.

Because Couchbase Server deploys an independent cluster for each data center, there are no limitations to how much data can be written to each data center. The clusters can be scaled out by adding more nodes with all nodes accepting writes.

If MongoDB is deployed to multiple data centers by placing a primary node—a node that can perform writes—in each data center. That means the amount of data that can be written to each data center is limited by a single node. It is possible to add more primary nodes by sharding the cluster further, but it adds more maintenance, more complexity, and more overhead.

Impact on performance

Couchbase Server can increase both read and write performance with geo-replication. It enables all applications to be able to read and write all data to nearest data center. However, MongoDB is limited to increasing read performance. While it can enable local reads, it still relies on remote writes and remote writes suffer from high latency.

Detailed additional information on this topic and for the summary can be found in Appendix A.

6. Caching

A memory cache improves database performance by keeping frequently accessed data in system memory for quick retrieval. Requests are received by the server and are checked against the memory cache. With NoSQL databases, issues of how data is stored in memory, the granularity of the data, and how the database communicates with the application and presentation levels are important to consider.

6.1 MongoDB

MongoDB's MMAP storage engine uses employs memory-mapped files that rely on the operating system page cache, mapping 4096-byte pages of file data to memory. Its WiredTiger engine (part of MongoDB since the acquisition of WiredTiger in December 2014 has an uncompressed block cache as well as a compressed OS page cache.

Writes remain in memory until MongoDB calls "fsync." With the default storage engine, MMAP, that is every 100 ms for the journal and 60 s for the data. With the WiredTiger storage engine, that is every 60 s or 2 GB of data. If called data is not in memory, then MongoDB uses page faults (soft or hard) to read data.

6.2 Couchbase Server

Couchbase Server employs a managed object cache to cache individual documents. Writes go directly to the cache, with a pool of threads continuously flushing new and modified documents to disk. If data is not in memory, Couchbase reads from disk.

An eviction process ensures there is room to read from disk and place in memory. Couchbase has two options here: only evict documents and retain all metadata (optimal when low latency is required), or evict documents and metadata (optimal with massive data sets and super-low latency access to working docs is required).

6.3 Impact on Performance

MongoDB caching is coarse-grained, caching blocks of documents. Removing a block can therefore remove many documents from memory. Performance drops when data is not in the OS page cache, and it's possible to cache data twice (in block cache and in WiredTiger's OS page cache.

In contrast, Couchbase Server caching is fine-grained, caching individual documents. Users can remove or add single documents to the cache, and the database does not rely on the OS page cache. Couchbase Server supports the memcached protocol, and there is no need for a separate caching tier.

Thus, there are options to optimize performance for small and large datasets with differing access patterns. In addition, because Couchbase Server does not rely on the OS page cache, maintenance operations that scan the entire data set will not impact the cache. For example, compaction will not result in frequently read item being replaced with random items.

7. Conclusion

This paper examined architecture and why it is important, from an enterprises' initial analysis of NoSQL database options through its actual deployment and day-to-day usage. Architecture clearly has a significant and direct impact on performance, scalability, and availability. The two databases under examination here may seem similar at first glance, as they are both NoSQL document-oriented databases designed to handle a variety of unstructured data.

However, it is clear that depending on an enterprise's unique needs—including the size of its datasets, how quickly data must be available and what percentage of the data must be available quickly, and perhaps most important, the ability for the database to scale with the enterprise—there is likely a mission-critical need for an enterprise to choose the database that best fits the complexities of its needs.

At Altoros, we not only encourage companies to contact the respective technology vendors directly with a long list of questions and requirements, but we are also available for deeper analysis of specific scenarios and use cases.

8. Appendix A: Notes on Main Topics

8.1 Notes on Topology

When mongos is installed on the MongoDB client machine, the two NoSQL solutions MongoDB and Couchbase begin to look more similar at the high level. In both deployments, for example, clients appear to be aware of cluster topology.

However, the Config Servers and Replica Sets for MongoDB require a much more complicated installation and configuration procedure than a single-package installation for Couchbase Server.

8.2 Notes on Partitioning

MongoDB

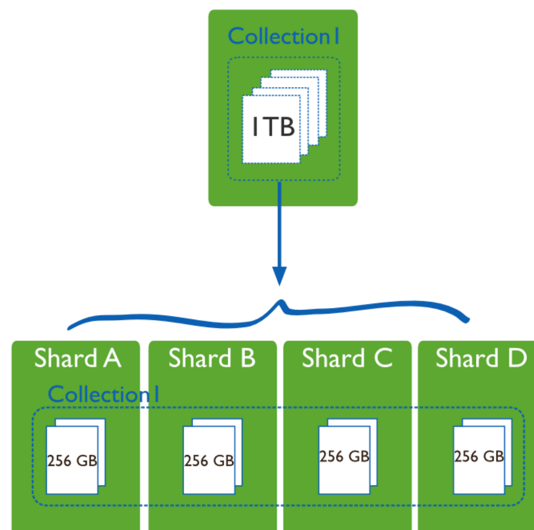
Logical. MongoDB documents are logically grouped into *collections*. A collection can be considered as an equivalent for an RDBMS table. Collections exist within *databases,* which are the physical containers for data (sets of files on the filesystem). Collections do not enforce any schema.

Besides an ordinary type of document collection, MongoDB also provides *capped collections*—fixed-size collections that support high-throughput operations that insert and retrieve documents based on insertion order. Capped collections work like circular buffers: once a collection fills its allocated space, it makes room for new documents by overwriting the oldest documents in the collection. Capped collections are actually the only way to limit physical resources dedicated to a particular collection. The collection can be limited both in the maximum number of records and total data set size.

Since MongoDB 3.0 different storage engines are available as a per-collection option for ordinary types. The current common options are MMAPv1 (original MongoDB storage engine) and WiredTiger.

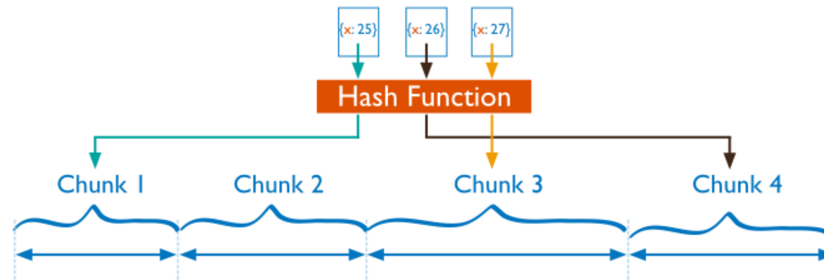
Physical. *Sharding* is a horizontal partitioning method for storing data across multiple machines so that different documents belong to different partitions, or *shards*. This method of distributing documents among servers allows larger data sets and higher write and read throughput for MongoDB. In practice, each shard is an independent database. It can be used separately from other shards or collectively form a single logical database responsible for the whole data set.

Sharding partitions a collection's data by an indexed field or indexed compound field that must exist in every document in the collection. This field is called shard key. MongoDB divides the whole set of shard-key values into chunks and distributes them evenly across the shards. The default chunk size is 64 MB. To divide the set of documents into chunks, MongoDB uses either range-based or hash-based partitioning.

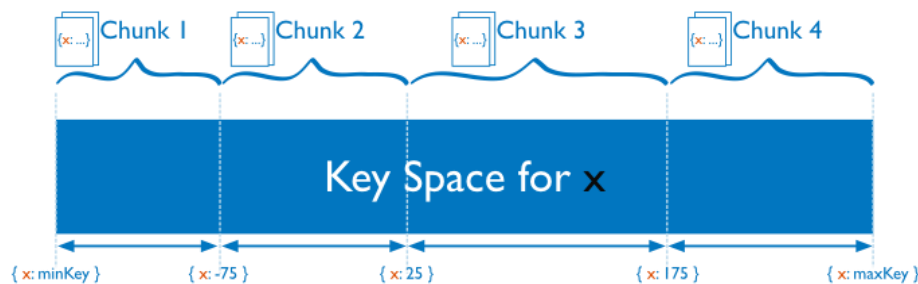


MongoDB supports three types of partitioning.

Hash partitioning computes hashes of the shard-keys and uses these hashes to create chunks. Two documents with close shard-key values are unlikely to appear in the same chunk. Hash-based sharding ensures a random distribution of documents in a collection across the shards. This type of MongoDB sharding is used to insure better CRUD operations balancing.



In *range-based sharding* (or partitioning), MongoDB divides the data set into shard-key ranges. Two documents with close shard-key values are very likely to appear in the same chunk.



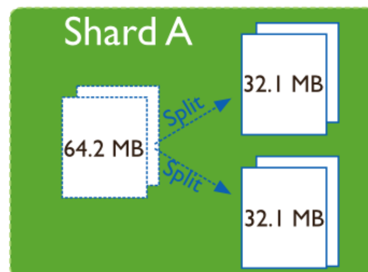
Range partitioning supports efficient range queries. Given a range query on the shard key, the router can easily determine which chunks overlap that range and route the query to only those shards that contain these chunks.

However, range partitioning result in an uneven distribution of data which negate some sharding benefits. For example, if the shard-key is a linearly increasing field, such as date, all the queries in a particular date range will hit a single chunk. The chunk will became a hot spot and eventually a bottleneck limiting the query performance within a single server responsible for that chunk. Hash partitioning, in contrast, ensures a uniform distribution of data paying for that with inefficient range queries. The random distribution makes a range query be unable to target a few shards, but more likely target every shard in order to return a result because the target range is evenly distributed across all the partitions.

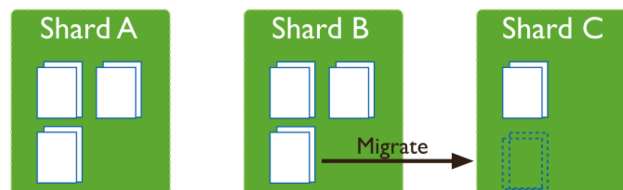
MongoDB also supports *Tag Aware Sharding* (or partitioning). Administrators create and associate tags with ranges of the shard key, and then assign those tags to the shards. Then, the balancer migrates tagged data to the appropriate shards and ensures that the cluster always enforces the distribution of data that the tags describe.

Process. The addition of new data or new servers can result in data distribution imbalances within the cluster. For example, a particular shard might contain significantly more chunks than another shard, or the size of a chunk is significantly greater than other chunk sizes.

MongoDB ensures a balanced cluster using two background process: splitting and the balancer. Splitting is a background process that keeps chunks from growing too large. When a chunk grows beyond a specified chunk size, MongoDB splits the chunk in half. Inserts and updates trigger splits.



The balancer is a background process that manages chunk migrations. The balancer can run from any the query routers in a cluster (mongos). When the distribution of a sharded collection in a cluster is uneven, the balancer process migrates chunks from the shard that has the largest number of chunks to the shard with the least number of chunks until the collection balances.



The shards manage chunk migrations as a background operation between an origin shard and a destination shard. During a chunk migration, the destination shard is sent all the current documents in the chunk from the origin shard. Next, the destination shard captures and applies all changes made to the data during the migration process. Finally, the metadata regarding the location of the chunk onconfig server is updated.

If there is an error during the migration, the balancer aborts the process, leaving the chunk unchanged on the origin shard. MongoDB removes the chunk's data from the origin shard after the migration completes successfully.

Couchbase Server

Logical. From the client application point of view, *buckets* in Couchbase Server can be considered as an analog for collections in MongoDB. Both are logical groupings of documents in a cluster.

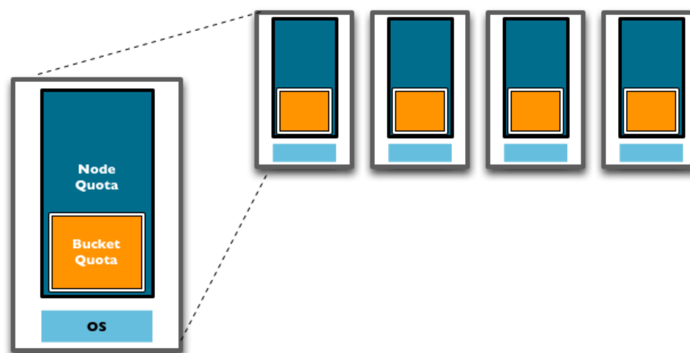
At the same time, buckets are the logical grouping of cluster physical resources, as well. They provide a mechanism for organizing, managing, and analyzing data storage resources. Quotas for RAM and disk usage are configurable per bucket, so that resource usage can be managed across the cluster.

RAM is allocated in the following configurable quantities: Server Quota and Bucket Quota.

The Server Quota is the RAM that is allocated to the server when Couchbase Server is first installed. This sets the limit of RAM allocated by Couchbase for caching data for all buckets and is configured on a per-node basis. The Server Quota is initially configured in the first server in your cluster, and the quota is identical on all nodes. For example, if you have 10 nodes and a 16 GB Server Quota, there is 160 GB RAM available across the cluster. If you were to add two more nodes to the cluster, the new nodes would need 16 GB of free RAM, and the aggregate RAM available in the cluster would be 192 GB.

The Bucket Quota is the amount of RAM allocated to an individual bucket for caching data. Bucket Quotas are configured on a per-node basis, and is allocated out of the RAM defined by the Server Quota. For example, if you create a new bucket with a Bucket Quota of 1 GB, in a 10-node cluster there would be an aggregate bucket quota of 10 GB across the cluster. Adding two nodes to the cluster would extend your aggregate bucket quota to 12 GB.

Quotas can be modified manually on a running cluster, and administrators can reallocate resources as usage patterns or priorities change over time.

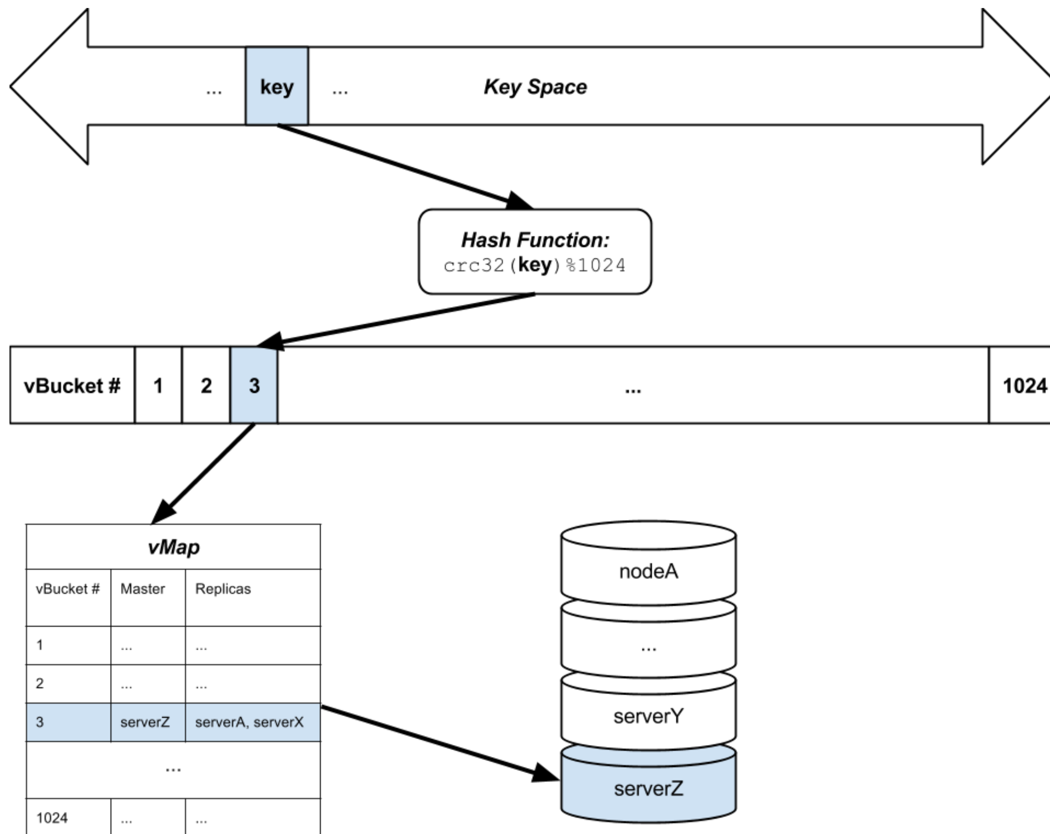


Physical. Foundational mechanism for Couchbase Server horizontal data partitioning is called vBuckets. The mechanism provides both partition tolerance and consistency for the system, creates a level of indirection between a user-defined document key and the actual address where the content is stored.

JSON-document is a basic unit in which data is stored in Couchbase Server (as in MongoDB and any other document-oriented database). Each document is associated with a key. The overall key space in a Bucket is divided with a hash function into 1,024 logical storage units, hidden from database users. These units are "virtual buckets," and referred to as vBuckets or Partitions.

vBuckets are distributed across machines within the cluster and a map called vMap is built. The vMap is shared among servers in the cluster as well as connected client library instances. vMap changes when the total number of cluster nodes changes, for example, as the cluster scales out or when a node fails.

With the help of the vBuckets mechanism, a client always knows which node to communicate to perform a CRUD operation on a particular key and related document. This direct access enables clients to communicate with the node storing the data instead of using a proxy or a balancer. As a result the physical topology is abstracted from the logical partitioning of data.



As we said before, client has an up-to-date mapping of a document key to a particular server mastering that key. Every key belongs to a particular partition—vBucket. Mapping function inside the client library is used to calculate the target vBucket. That function is a usual hashing function that takes a document key as input and returns the vBucket identifier. After the vBucket identifier has been calculated, the vMap table is used to get the server currently mastering the target vBucket. The vMap table also holds the information about the vBuckets replicas, so the client can perform replica read if needed. After the client library has finally calculated the target server, it can send the client operation directly to that server.

Process. As you store data into the Couchbase Server cluster, you may need to alter the number of nodes in the cluster to cope with changes in application load, RAM, disk I/O and network performance requirements.

Data is stored within Couchbase Server through the distribution method that is provided by the vBucket structure. When the Couchbase Server cluster is expanded or shrunk, the information stored in the vBuckets is redistributed among the available nodes and the corresponding vBucket map is updated to reflect the new structure. This process is called rebalancing.

The purpose of any rebalance is to bring up a cluster to a healthy state where the configured nodes, buckets, and replicas match the current state of the cluster. The operation is performed to expand or reduce the size of a cluster, to react to fail-overs, or to temporarily remove nodes for various upgrades.

Rebalancing is a deliberate process that you need to initiate manually when the structure of your cluster changes. It changes the allocation of the vBuckets used to store the information and then physically moves the data between the nodes to match the new structure.

The rebalance process is managed through a specific process called orchestrator, which examines the current vBucket map, then combines that information with the node additions and removals in order to create a new vBucket map. It only starts the process while the nodes themselves are responsible for actually performing the movement of data between the nodes.

The rebalancing process can take place while the cluster is running and servicing requests. Clients using the cluster read and write to the existing structure with the data being moved in the background among nodes. Once the moving process has been completed, the vBucket map is updated and communicated to the clients and the proxy service (Moxi). Because the cluster stays up and active throughout the entire rebalancing process, clients can continue to store and retrieve information and do not need to be aware that a rebalance operation is taking place.

As a result, the distribution of data across the cluster is rebalanced, or smoothed out, so that the data is evenly distributed across the database. Rebalancing takes into account the data and replicas of the data required to support the system.

There are four primary reasons to perform a rebalance operation:

- Adding nodes to expand the size of the cluster
- Removing nodes to reduce the size of the cluster
- Reacting to a failover situation, where you need to bring the cluster back to a healthy state
- Temporary removing one or more nodes to perform a software, operating system, or hardware upgrade

(In contrast, MongoDB can run its balancer even if the number of cluster nodes stays the same but the data set changes.)

Couchbase Server’s rebalance operation can be stopped at any time and incrementally continued later. This is triggered from the Admin Web UI or from the CLI.

8.3 Notes on Couchbase Replication

When a client application writes data to a node, that data is placed in a replication queue, then a copy is sent to another node. The replicated data is then available in RAM on the second node and placed in a disk write queue to be stored on disk at the second node.

The second node also simultaneously handles both replica data and incoming writes from a client. It puts both replica data and incoming writes into a disk write queue. If there are too many items in the disk write queue, this second node can send a backoff message to the first node. The first node will then reduce the rate at which it sends items to the second node for replication. (This is sometimes necessary if the second node is already handling a large volume of writes from a client application.)

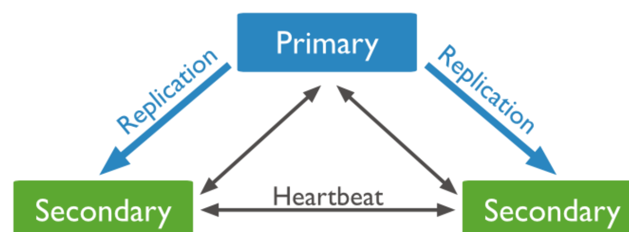
Couchbase Server provides both strong consistency guarantees with default master reads, and eventual consistency by special replica read API calls. Replica read is a very special and rare case for handling node failure situations when a node mastering a particular key could be unavailable. Both read and write load are evenly distributed across Couchbase Server cluster by default with the vBucket system.

8.4 Notes on Availability

MongoDB

Within the replica set, members are interconnected with each other to exchange heartbeat messages. A crashed server with a missing heartbeat will be detected by other members and removed from the replica set membership. After the dead secondary recovers, it can rejoin the cluster by connecting to the primary, then catch up to the latest update.

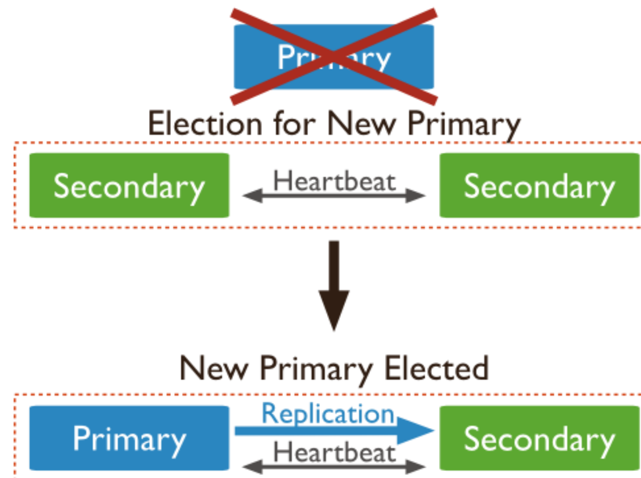
If a crash occurs over a lengthy period of time, where the change log from the primary doesn't cover the whole crash period, then the recovered secondary needs to reload the whole data from the primary as if it was a brand new server.



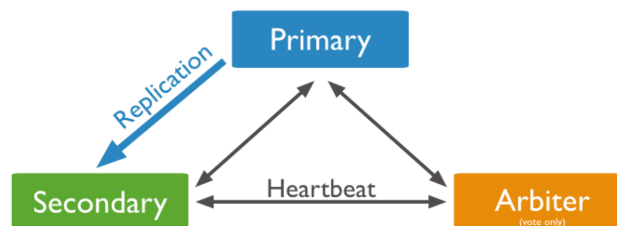
In case of a primary DB crash, a leader election protocol will be run among the remaining members to nominate the new primary, based on many factors such as the node priority, node uptime, etc. After getting the majority vote, the new primary server will take its place.

Note that due to async replication, the newly elected primary DB doesn't necessary have all the latest updated from the crashed DB.

When a primary does not communicate with the other members of the set for more than 10 seconds, the replica set will attempt to select another member to become the new primary. The first secondary that receives a majority of the votes becomes the primary.



One may need to add an extra mongod instance to a replica set as an arbiter. Arbiters do not maintain a data set, but only exist to vote in elections. If your replica set has an even number of members, add an arbiter to obtain a majority of votes in an election for primary. Arbiters do not require dedicated hardware.



Config Server. Config servers are special mongod instances that store the *metadata* for a sharded cluster. Config servers use a two-phase commit to ensure immediate consistency and reliability. Config servers *do not* run as replica sets. All config servers must be available to deploy a sharded cluster or to make any changes to cluster metadata.

The metadata reflects the state and organization of the sharded data sets and system. It includes the list of chunks on every shard and the ranges that define the chunks.

MongoDB writes data to the config server in the following cases:

- To create splits in existing chunks
- To migrate a chunk between shards

MongoDB reads data from the config server data in the following cases:

- A new mongos starts for the first time, or an existing mongos restarts.
- After a chunk migration, the mongos instances update themselves with the new cluster metadata.

MongoDB also uses the config server to manage distributed locks.

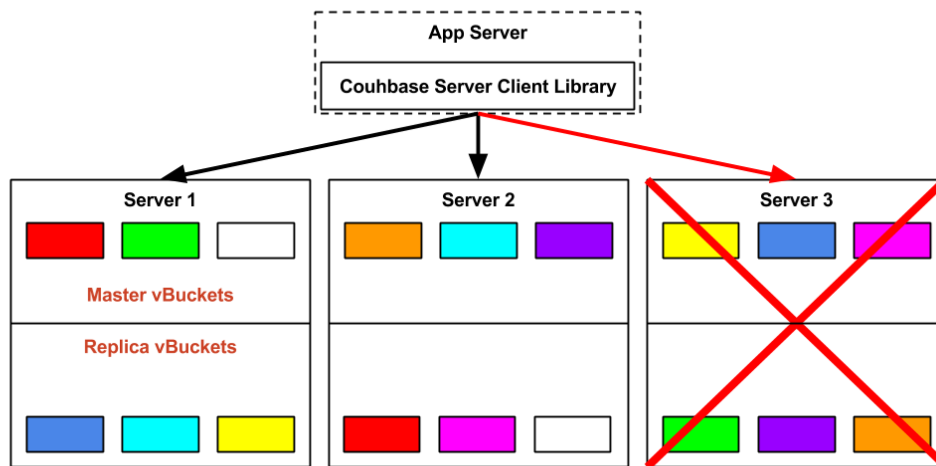
Elections. Replica set members send heartbeats (pings) to each other every two seconds. If a heartbeat does not return within 10 seconds, the other members mark the delinquent member as inaccessible. As noted above, when the primary becomes inaccessible the elections mechanism is triggered.

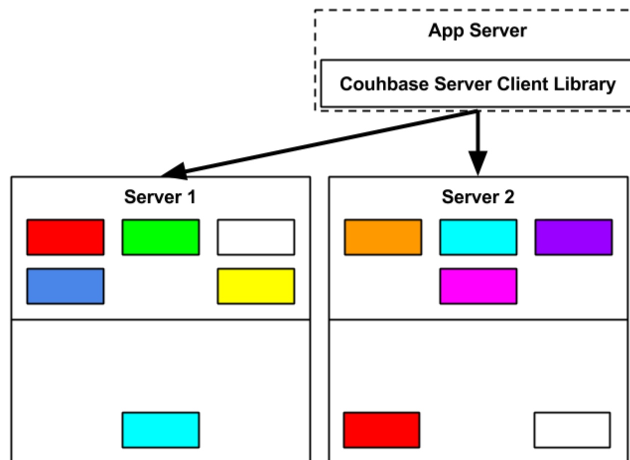
Elections are essential for independent operation of a replica set, but they take time to complete. While an election is in process, the replica set has no primary and cannot accept writes. All remaining members become read-only. If a majority of the replica set is inaccessible or unavailable, the replica set cannot accept writes and all remaining members become read-only.

Couchbase Server

Failover is the process in which a node in a Couchbase Server cluster is declared as unavailable, and a replica vBucket is promoted to an active state to start serving write operations. In case of node failure or cluster maintenance, the failover process contacts each server that was acting as a replica and updates the internal table that maps client requests for documents to an available Couchbase Server.

Failover can be performed manually or automatically using the built-in automatic failover process. Auto failover acts after a preset time, when a node in the cluster becomes unavailable.





On the picture above, one server goes down and three replica vBuckets are promoted for mastering the data after a failover process is finished. The load balance in this case is broken, as one server is mastering more data and gets more requests from clients. The rebalance process must be triggered in this scenario.

Cluster Wide Services. Among Cluster Managers, one is elected as master. The master node starts a few cluster-wide services that have to be run once across the cluster. These services are registered in the erlang global naming facility, which ensures that there's only one process registered under given name in a cluster.

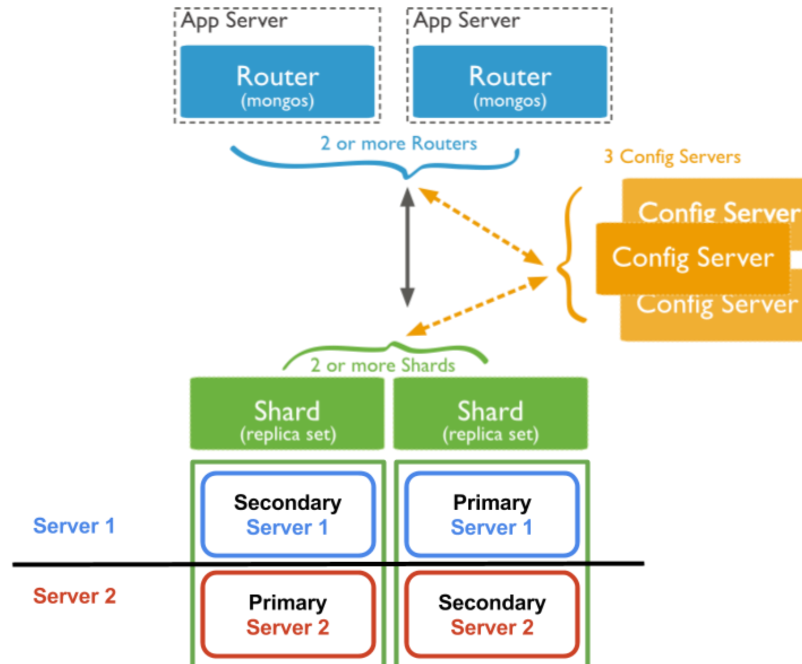
Among these services, one called “ns_orchestrator” is a central coordination service. It does: bucket creation, bucket deletion, "janitoring" (activating vBuckets and spawning/restoring replications to match vbucket map, normally after node restart), rebalance, and failover.
With the help of several spawned child processes, ns_orchestrator performs vBucket moves - one per time constantly updating the vMap. The rebalance process is designed so it can be stopped any time and continued later from that point.

Elections. Couchbase Server has elections for cluster-wide services inside Cluster Manager.

However, true singleton cluster-wide service is a general problem of distributed systems. Network partitioning can happen and a split-brain situation can occur.

Summary. To achieve a MongoDB deployment closely equivalent to Couchbase Server, one should configure replica sets to have a primary mongod on each of the servers in the MongoDB cluster, and have a number of secondaries same as replica factor used in Couchbase Server cluster.

The same shard must contain primary and secondaries located on different servers. This way one can configure MongoDB to have the load balanced evenly across all the cluster nodes, as in Couchbase Server. The picture below illustrates a configuration with two servers and replica factor one (primary mongod + one secondary).



8.5 Notes on Read Path

MongoDB

The current MongoDB Manual provides excellent, detailed information on this topic. We will quote extensively from it here.

"In MongoDB, queries select *documents* from a single *collection*. Queries specify criteria, or conditions, that identify the documents that MongoDB returns to the clients. A query may include a *projection* that specifies the fields from the matching documents to return. The projection limits the amount of data that MongoDB returns to the client over the network."

"For query operations, MongoDB provides a db.collection.find() method. The method accepts both the query criteria and projections and returns a *cursor* to the matching documents. You can optionally modify the query to impose limits, skips, and sort orders."

Source: http://docs.mongodb.org/manual/core/read-operations-introduction/

"This method queries a collection and returns a *cursor* to the returning documents. To access the documents, you need to iterate the cursor."

Cursor behavior

- **Closure of inactive cursors**. “By default, the server will automatically close the cursor after 10 minutes of inactivity or if client has exhausted the cursor”;
- **Isolation**. “because the cursor is not isolated during its lifetime, intervening write operations on a document may result in a cursor that returns a document more than once if that document has changed”;
- **Batches**. “The MongoDB server returns the query results in batches. Batch size will not exceed the *maximum BSON document size*. For most queries, the *first* batch returns 101 documents or just enough documents to exceed 1 megabyte. Subsequent batch size is 4 megabytes. <...> For queries that include a sort operation *without* an index, the server must load all the documents in memory to perform the sort before returning any results. As you iterate through the cursor and reach the end of the returned batch, if there are more results, cursor.next() will perform a getmore operation to retrieve the next batch.”
- **Information**. “The db.serverStatus() method returns a document that includes a metrics field. The metrics field contains a cursor field with the following information: number of timed out cursors since the last server restart, number of open cursors with the option DBQuery.Option.noTimeout set to prevent timeout after a period of inactivity, number of “pinned” open cursors, total number of open cursors.”

  http://docs.mongodb.org/manual/core/cursors/

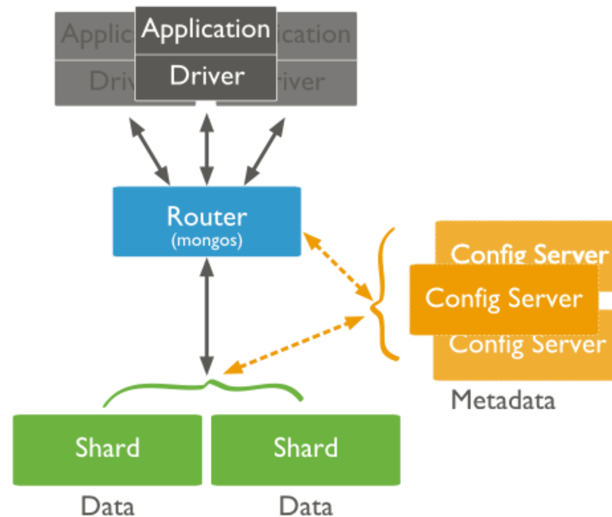
“<...> an index on the queried field or fields can prevent the query from scanning the whole collection to find and return the query results. In addition to optimizing read operations, indexes can support sort operations and allow for a more efficient storage utilization.”

http://docs.mongodb.org/manual/core/query-optimization/

Be aware for the following limitation while “Covering a Query”:

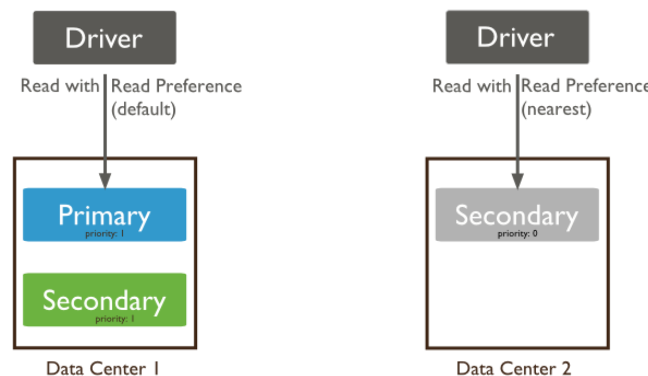
http://docs.mongodb.org/manual/core/query-optimization/#covering-a-query
http://docs.mongodb.org/manual/core/query-optimization/#limitations

“For a sharded cluster, applications issue operations to one of the mongos instances associated with the cluster. Queries to sharded collections should include the collection’s *shard key*. When a query includes a shard key, the mongos can use cluster metadata from the *config database* to route the queries to shards. If a query does not include the shard key, the mongos must direct the query to *all* shards in the cluster. These *scatter gather* queries can be inefficient.”



More: http://docs.mongodb.org/manual/core/distributed-queries/

"*Replica sets* use *read preferences* to determine where and how to route read operations to members of the replica set. By default, MongoDB always reads data from a replica set's *primary*. You can configure the *read preference mode* on a per-connection or per-operation basis <...>"



"Read operations from secondary members of replica sets are not guaranteed to reflect the current state of the primary, and the state of secondaries will trail the primary by some amount of time. Often, applications don't rely on this kind of strict consistency, but application developers should always consider the needs of their application before setting read preference."

Read Isolation. "MongoDB allows clients to read documents inserted or modified before it commits these modifications to disk, regardless of write concern level or journaling configuration. As a result, applications may observe two classes of behaviors:

- For systems with multiple concurrent readers and writers, MongoDB will allow clients to read the results of a write operation before the write operation returns.
- If the mongod terminates before the journal commits, even if a write returns successfully, queries may have read data that will not exist after the mongod restarts.

Other database systems refer to these isolation semantics as *read uncommitted*. For all inserts and updates, MongoDB modifies each document in isolation: clients never see documents in intermediate states."

More here: http://docs.mongodb.org/manual/core/write-concern/#read-isolation

Couchbase Server

Couchbase operates by having the Partition Map (vMap) client send the read request directly to a particular node holding the needed document. More precisely, the request goes to the node's Data Manager. The Data Manager routes the request to the corresponding ep-engine responsible for the vBucket.

The ep-engine will look up the document id from the in-memory hash table. If the document content is found in cache (stored in the value of the hash table), it will be returned. Otherwise, a background disk fetch task will be created and queued into the Batch Reader. Then the Batch Reader gets the value from the underlying storage engine and populates the corresponding entry in the Partition Hash Table. The notification thread notifies the disk fetch completion to the memcached pending connection, so that the memcached worker thread can revisit the engine to process a get request.

8.6 Notes on Durability

Durability requirements describe the guarantee that the database provides when reporting on the success of a write operation. The strength of the requirements determine the level of guarantee.

When inserts, updates, and deletes have weak requirements, write operations return quickly. In some failure scenarios such operations may not persist.

In contrast, with stronger write durability requirements, clients wait after sending a write operation for the database to confirm the persistence.

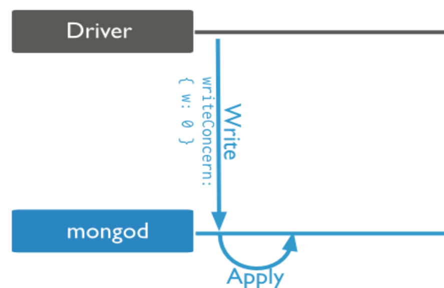
MongoDB

Write durability requirements referred as *write concern* in MongoDB. Default write concerns may be adjusted for a particular replica set or the applications can determine the acceptable write concern level on per-call basis.

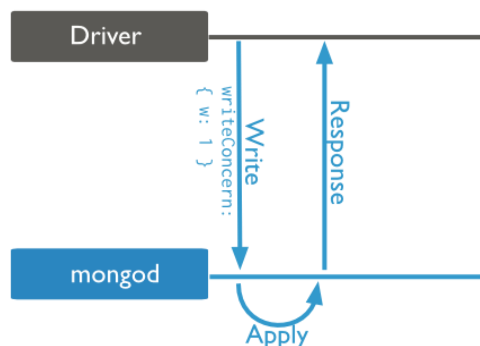
The client can include the *w* option to specify the required number of mongod acknowledgments before returning. The *j* option to require writes to the journal before returning. The client submits the options within each call and mongos instances pass the write concern on to the shard.

When a single mongod returns a successful *journaled write concern*, the data is fully committed to disk and will be available after mongod restarts. For replica sets, write operations are durable only after a write replicates to a *majority* of the voting members of the set and commits to the primary's journal.

Unacknowledged - w: 0 Durability is not required. This option is not recommended. MongoDB does not acknowledge the receipt of write operations. *Unacknowledged* is similar to *errors ignored*; however, drivers will attempt to receive and handle network errors when possible. If the client disables basic write operation acknowledgment but require journal commit acknowledgment, the journal commit prevails, and the server will require that mongod acknowledge the write operation.

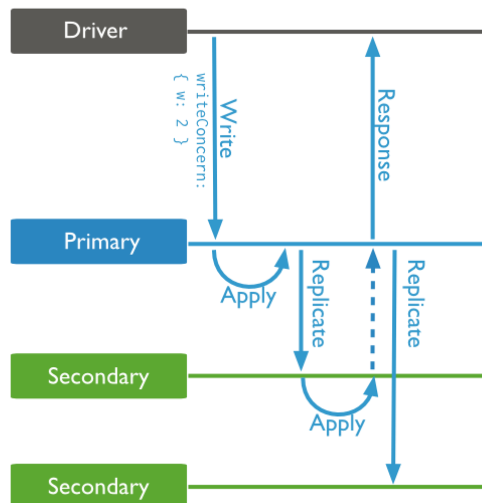


Acknowledged - w: 1 With a receipt *acknowledged* write concern, the standalone mongod or the *primary* in a replica set confirms that it received the write operation and applied the change to the in-memory view of data. *Acknowledged* write concern allows clients to catch network, duplicate key, and other errors. *Acknowledged* write concern still does *not* confirm that the write operation has persisted to the disk system.



MongoDB uses the *acknowledged* write concern by default.

Replica Acknowledged w: >1 With *replica acknowledged* write concern, the client can guarantee that the write operation propagates to additional members of the replica set. For example, w: 2 indicates acknowledgements from the primary and at least one secondary.

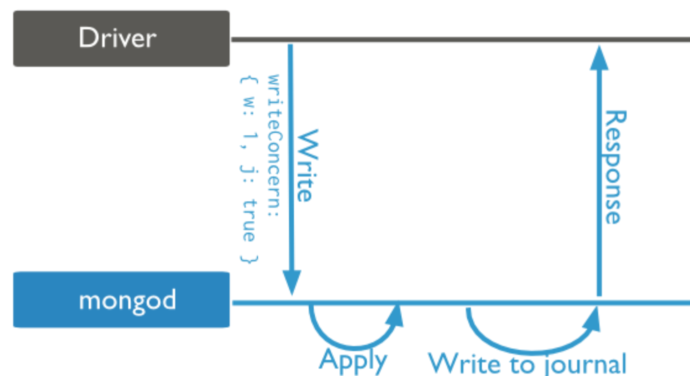


The client can include a timeout threshold for a write concern. This prevents write operations from blocking indefinitely if the write concern is unachievable. For example, if the write concern requires acknowledgement from four members of the replica set and the replica set has only three available, the operation blocks until those members become available. This scenario could cause potentially infinite wait time on the client side.

w: “majority”
Confirms that write operations have propagated to the mongods the majority of the configured replica set. This allows the client to avoid hard-coding assumptions about the size of the replica set into the client application.

Journaled With a *journaled* write concern, MongoDB acknowledges the write operation only after committing the data to the *journal*. This write concern ensures that MongoDB can recover the data following a shutdown or power interruption. With a *journaled* write concern, write operations must wait for the next journal commit (2–200 ms).

Requiring *journaled* write concern in a replica set only requires a journal commit of the write operation to the *primary* of the set regardless of the level of *replica acknowledged* write concern.



Couchbase Server

Because Couchbase Server client is topology-aware, with a CRUD operation it goes straight to the node mastering the particular document.

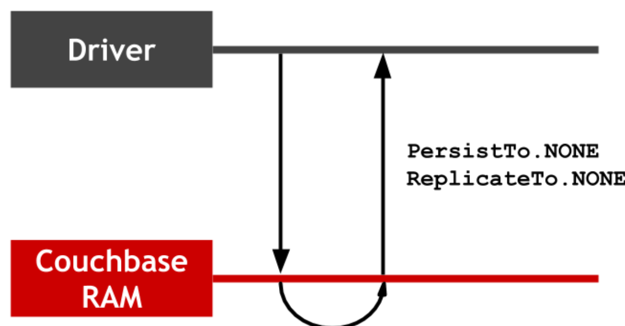
In its durability requirements mechanism, the client can submit special parameters called *ReplicateTo* and *PersistTo* within an each insert or update client library call. The parameters can take different values like *MASTER*, *ONE,* or *TWO*.

The first parameter—*ReplicateTo*—specifies the number of cluster nodes that must have the document in their managed object cache before the operation acknowledgement will happen.

The second parameter—*PersistTo*—specifies the number of cluster nodes that must have the document persisted on disk before the operation acknowledgement.

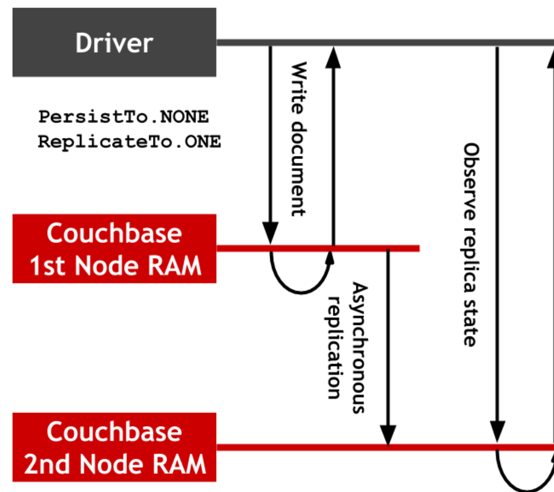
The internal implementation first performs a regular write operation, and afterwards starts polling the specific affected cluster nodes for the state of the document. If something fails during this operation, the original operation might have succeeded nonetheless.

ReplicateTo.NONE This durability requirement is the default and guarantees that the mutation is in the Partition Hash Table of the appropriate vBucket mastering the modified document.

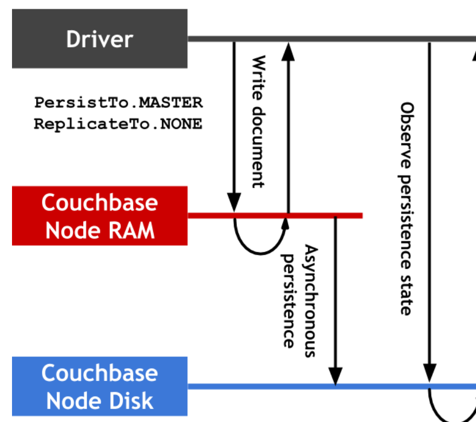


Replication and on-disk persistence are asynchronous, and the client can check the status of document manually later by a special Observe API that will return replication and persistence status.

ReplicateTo.ONE (TWO, THREE) ReplicateTo.ONE durability requirement option guarantees that the data modification is in memory of both the nodes with master and replica vBuckets. It is implemented inside the driver via hidden periodical checking of the state of the modification on the appropriate cluster nodes.



PersistTo.MASTER With *PersistTo.MASTER* durability requirement, the client application notifies Couchbase Server SDK that it needs to acknowledge write operation only after the data modification is persisted on disk. SDK Driver is responsible for periodically checking the persistence status and forming synchronous or asynchronous client application notification.



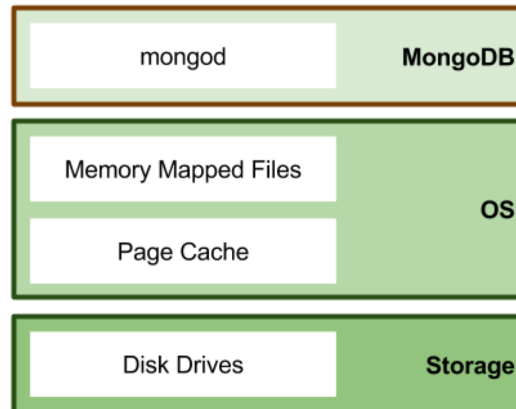
PersistTo.TWO (THREE, FOUR) It is possible to combine different *PersistTo* and *ReplicateTo* values to achieve any sensible durability requirements. In combination with rack-aware cluster configuration, *PersistTo* parameter values can fulfill extremely rigorous application requirements for durability.

9. Appendix B. Notes on Memory

9.1 MongoDB

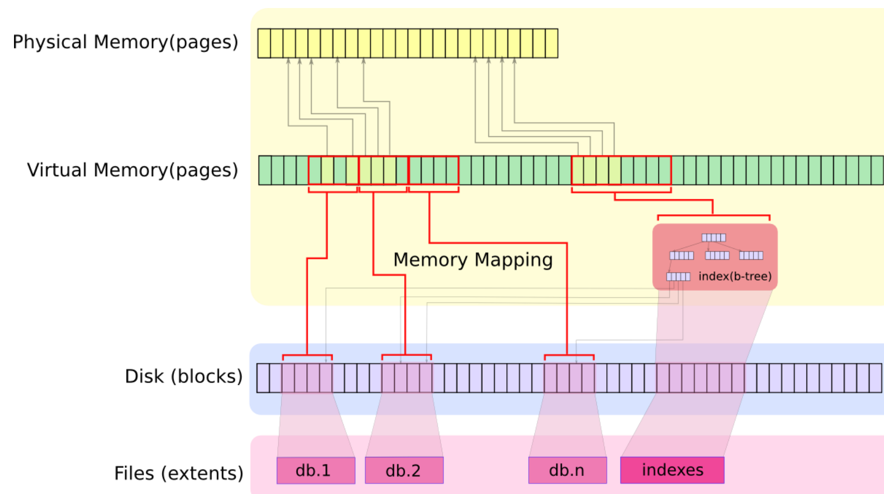
MongoDB uses memory-mapped files for its data storage. Since MongoDB 3.0 there are storage backend options:

- MMAPv1, default and original backend based on the memory-mapping technique
- WiredTiger, a modern storage engine with LSM and B+Trees inside



Mongod process with MMAPv1 does not use a separate caching data structure, but performs I/O operations directly on files mapped into its virtual memory. Memory management responsibility is left to the OS, which itself determines which data chunks need to be present in RAM or could be evicted on disk.

Files are mapped to memory using an mmap() system call. This is represented in the graphic below by solid red lines connecting the mapped disk blocks to the virtual memory pages.



As all the required files are mapped into the mongod's virtual memory, it can perform I/O operations on them with the usual pointer arithmetic. This virtual memory is then mapped a second time by the OS to physical memory.

This is done transparently when processes access virtual memory. If the memory page is not mapped, then a page fault occurs and the OS tries to load the page from disk.

However, if RAM is full, OS has to free some physical space for that page by reclaiming some resident pages. The precise page replacement algorithm is usually a variation of Least Recently Used algorithm (LRU) and can be very complex, but the effect is that some pages get potentially written to disk storage or just cleaned out.

There are several types of pages from the OS kernel point of view:

- locked—cannot be reclaimed.
- swappable—can be swapped.
- syncable—these can be reclaimed by persisting them on disk to the appropriate files or just removed if used for example from a file opened in read-only mode.
- discardable—can be just discarded.

Pages coming from MongoDB's data files are syncable. It is crucial for MongoDB to run exclusively on a server in order to use available RAM for system page cache and not interfere with other processes.

9.2 Couchbase Server

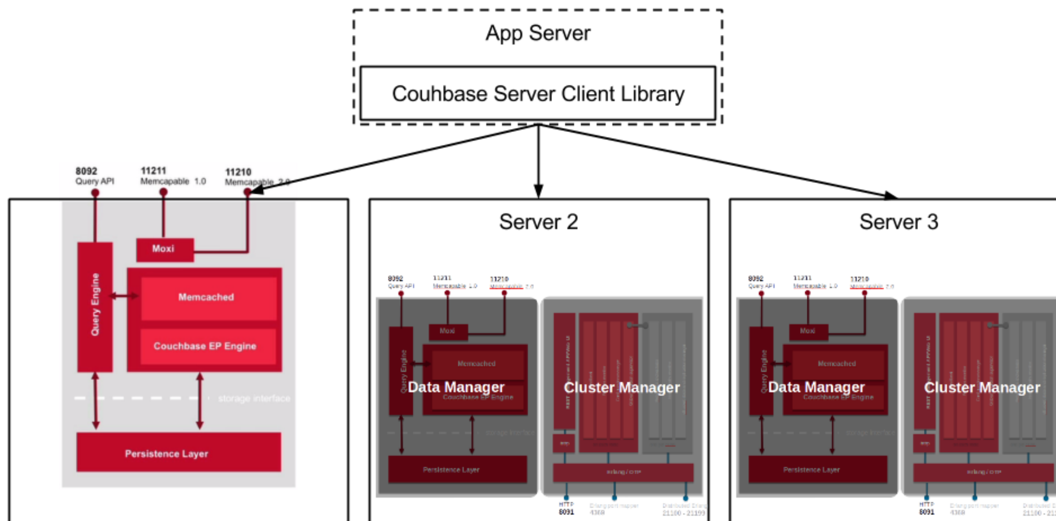
Couchbase Server has a built-in caching layer that acts as a central part of the server and provides very rapid reads and writes of data. Cluster automatically manages the caching layer and coordinates with disk space to ensure that enough cache space exists to maintain performance.

Couchbase automatically moves data from RAM to disk asynchronously, in the background, to keep frequently used information in memory and less frequently used data on disk. Couchbase constantly monitors the information accessed by clients and decides how to keep the active data within the caching layer. Data is ejected to disk from memory while the server continues to service active requests.

During sequences of high writes to the database, clients are notified if the server is temporarily out of memory until enough items have been ejected from memory to disk. The asynchronous nature and use of queues in this way enables reads and writes to be handled at a very fast rate, while removing the typical load and performance spikes that would otherwise cause a traditional RDBMS to produce erratic performance.

When the server stores data on disk and a client requests the data, an individual document ID is sent, then the server determines whether the information exists or not. Couchbase Server does this with metadata structures. The metadata holds information about each document in the database, and this information is held in RAM. This means that the server returns a “document ID not found” response for an invalid document ID, returns the data from RAM, or returns the data after being fetched from disk.

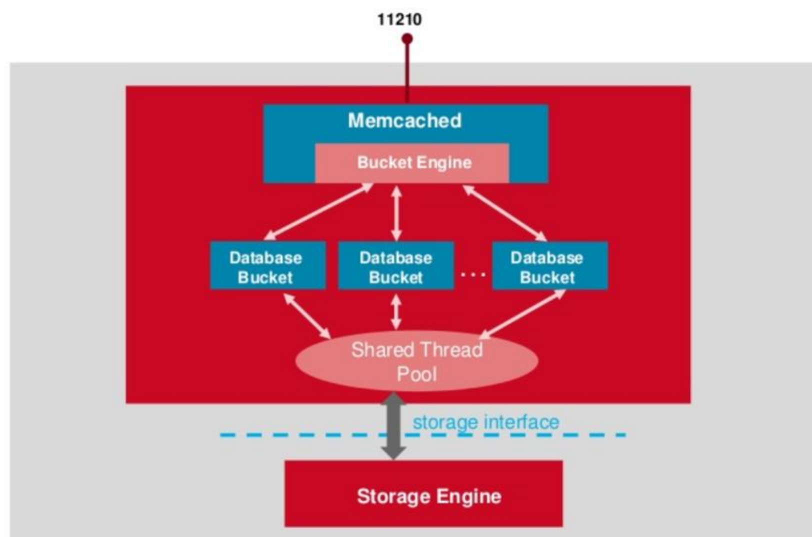
As previously mentioned, each Couchbase Server node has two major components: Data Manager and Cluster Manager.



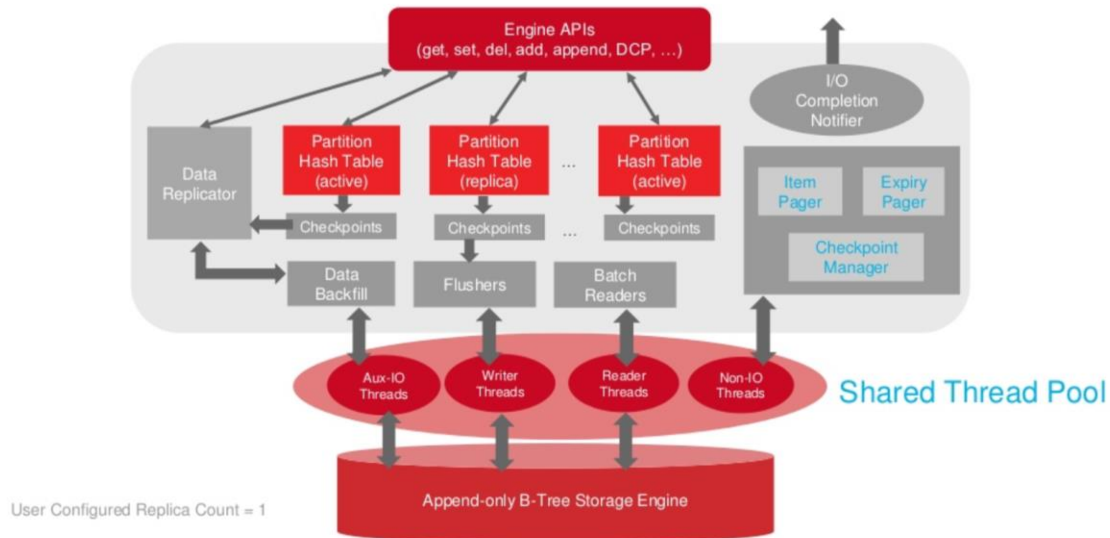
The Data Manager stores and retrieves data in response to data operation requests from applications. The majority of code in the Data Manager is written in C and C++. It exposes two ports to the network for key-value operations and one for the query API.

11211 is a traditional memcached port for non-smart client access. It can be proxied via Moxi, a memcached proxy for Couchbase. When the client sends a key-value request to port 11211, Moxi processes it and, if required, forwards it to the server currently servicing the key referenced by the request.

11210 is a port for smart vBucket-aware clients that know where to send a request for a particular key.



Within the Data Manager there is a multi-tenant Bucket Engine that is capable of serving multiple Buckets. Under the Buckets a Shared Thread Pool is used. It has three major types of threads: read, write and cache management. Storage Engine layer is represented by Couchstore (for Couchbase Server 3.x) and described in a section below.



As previously mentioned, Couchbase Server does data partitioning by the vBucket system. So in Couchbase Server documentation and sources a vBucket can be sometimes referred to as a Partition.

Each vBucket has its own Partition Hash Table, within in-memory cache. Each Partition Hash Table maintains its own Checkpoint data structure, which is a unified queue for both replication and persistence.

There is a per-bucket Flusher task that periodically visits all the queues and writes all the dirty items from them (documents) on disk through the shared thread pool mentioned before. The Batch Reader task provides item (document) fetching from the backing store for those items that are not resident in Partition Hash Tables.

The system as a whole is optimized for average latency (ms) and average throughput (ops/sec) for a number of operations within a special time window, not for an instant best latency for a single-access operation. Therefore read and write requests are grouped for replication or disk I/O inside those various tasks.

There's one more important task related to the in-memory Cache Management, formed by Item Pager, Expiry Pager, and Checkpoint Manager.

Item Pager is responsible for evicting non-dirty, not-recently-used items from the Partition Hash Tables, in order to keep only the most recently used items there. The selection algorithm is described

in Couchbase Server documentation and referred there as Not-Frequently(Recently)-Used Items - NF(R)U.

Expiry Pager is the task which collects expired items in the Partition Hash Tables and pushes them into corresponding Checkpoint queues to be replicated or persisted on disk. This task is run in the non-IO part of the shared thread pool.

At the high level are several settings to control Cache Management. In addition to Bucket memory quota—itself the quota for the caching layer—there are two watermarks the engine uses to determine when it is necessary to start persisting more data to disk. These are mem_low_wat and mem_high_wat. The item ejection process starts when the bucket memory usage goes beyond the high watermark, and stops the ejection when the memory usage drops below the low watermark.

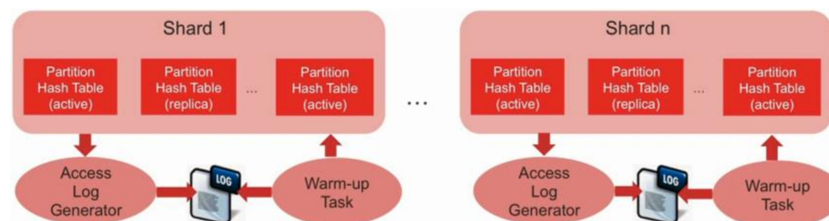
Since Couchbase Server 3.0 the Cache Management system has two major strategies for item ejection:

- Value-only ejection (the default) removes the data from cache but keeps all keys and metadata fields for non-resident items. When the value bucket ejection occurs, the item's value is reset.
- Full metadata ejection removes all data including keys, metadata, and key-values from cache for non-resident items. Full metadata ejection reduces RAM requirement for large buckets.

The ejection strategy is configured at the bucket level. The default value-only ejection strategy provides highly cache-oriented architecture, while full metadata ejection is fit for storing a larger number of documents.

Data Replicator is responsible for replicating of all the mutations to the corresponding replica vBuckets on the remote nodes. Data Replicator uses DCP streaming protocol since Couchbase Server 3.0.

Partition Hash Table (or vBucket's Hash Table) is a hash table chained with a linked list. Collisions are resolved by putting the items into a linked list. There are a number of locks that synchronize access to those Partition Hash Table lists (usually referred as buckets when speaking about hash table buckets).



The Access Log Generator daemon periodically captures cache state to be used later for the warm-up process needed in case of a node crash or restart.

10. Appendix C. Notes on Storage Engine

10.1 MongoDB

MongoDB stores data in the form of BSON (binary JSON) documents. All documents in MongoDB must be less than 16 MB, which is the maximum BSON document size.

Every document in MongoDB is stored in a *record* that contains the document itself and extra space, or *padding*, which allows the document to grow as the result of updates. All records are contiguously located on disk. When a document becomes larger than the allocated record, MongoDB must allocate a new record. New allocations require MongoDB to move a document and update all indexes that refer to the document, which takes more time than in-place updates and leads to storage fragmentation.

MongoDB supports multiple record allocation strategies that determine how mongod adds padding to a document when creating a record. These strategies support different kinds of workloads: the *power of 2 allocations* are more efficient for insert/update/delete workloads; while *exact fit allocations* is ideal for collections *without* update and delete workloads.

All records are part of a *collection*, which is a logical grouping of documents in a MongoDB database. The documents in a collection share a set of indexes, and typically these documents share common fields and structure.

In MongoDB the *database* is a group of related collections. Each database has a distinct set of data files and can contain a large number of collections. A single MongoDB deployment may have many databases.

Because documents in MongoDB may grow after insertion and all records are contiguous on disk, the padding can reduce the need to relocate documents on disk following updates. Relocations are less efficient than in-place updates, and can lead to storage fragmentation. As a result, all padding strategies trade additional space for increased efficiency and decreased fragmentation.

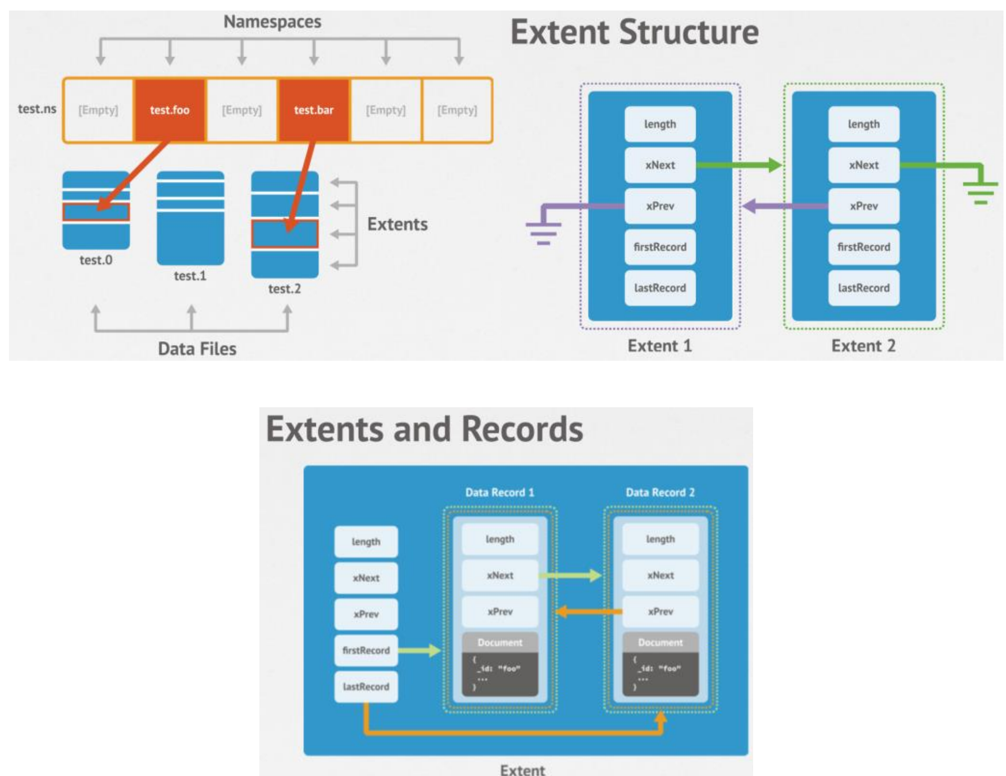
Memory-mapped files - MongoDB version < 3.0

https://github.com/mongodb/mongo/tree/v2.6/src/mongo/db/storage

Each document collection is stored in one namespace file that contains metadata information and multiple extent data files, with a doubling increasing size starting from 64 MB and ending up with 2 GB per file. Mongod uses aggressive pre-allocation—it will allocate disk space for the next file as soon as it will insert a fist document into current data file. Mongod file structure appears as follows:

```
drwxr-xr-x    136          Nov 19 10:12    journal
-rw-------    16777216     Oct 25 14:58    test.0
-rw-------    134217728    Mar 13  2012    test.1
-rw-------    268435456    Mar 13  2012    test.2
-rw-------    536870912    May 11  2012    test.3
-rw-------    1073741824   May 11  2012    test.4
-rw-------    2146435072   Nov 19 10:14    test.5
-rw-------    16777216     Nov 19 10:13    test.ns
```

The data structure uses doubly-linked-list extensively. Each collection of data is organized in a linked list of extents, each of which represents a contiguous disk space. Each extent points to a head/tail of another linked list of docs. Each doc contains a linked list to other documents as well as the actual data encoded in BSON format.



MongoDB uses *write ahead logging* to an on-disk *journal* to guarantee that it's able to quickly recover the *write operations* following a crash or other serious failure. Mongod records all modifications to the journal persisted to disk more frequently than the data files. The default and recommended maximum time can pass before MongoDB flushes data to the data files via a sync() system call is 60 seconds while journal commit interval is 100 ms.

MongoDB creates a journal subdirectory within the database directory, which is /data/db by default. The journal directory holds journal files, which contain write-ahead redo logs. The directory also holds a last-sequence-number file. A clean shutdown removes all the files in the journal directory. A dirty shutdown (crash) leaves files in the journal directory. These are used to automatically recover the database to a consistent state when the mongod process is restarted.

Journal files are append-only files and have file names prefixed with j._. When a journal file holds 1 gigabyte of data, MongoDB creates a new journal file. Once MongoDB applies all the write operations in a particular journal file to the database data files, it deletes the file, as it is no longer needed for recovery purposes.

Unless you write *many* bytes of data per second, the journal directory should contain only two or three journal files. You can use the storage.smallFiles run time option when starting mongod to limit the size of each journal file to 128 megabytes, if you prefer. To speed the frequent sequential writes that occur to the current journal file, you can ensure that the journal directory is on a different filesystem from the database data files.

Journaling adds three internal storage views to MongoDB:

1. The **shared view** stores modified data for upload to the MongoDB data files. The shared view is the only view with direct access to the MongoDB data files. When running with journaling, mongod asks the operating system to map your existing on-disk data files to the shared view virtual memory view. The operating system maps the files but does not load them. MongoDB later loads data files into the shared view as needed.

2. The **private view** stores data for use with *read operations*. The private view is the first place MongoDB applies new *write operations*. Upon a journal commit, MongoDB copies the changes made in the **private view** to the shared view, where they are then available for uploading to the database data files.

3. The **journal** is an on-disk view that stores new write operations after MongoDB applies the operation to the **private view** but before applying them to the data files. The journal provides durability. If the mongod instance were to crash without having applied the writes to the data files, the journal could replay the writes to the shared view for eventual upload to the data files.

MongoDB copies the write operations to the journal in batches called group commits. These “group commits” help minimize the performance impact of journaling, since a group commit must block all writers during the commit. See commitIntervalMs for information on the default commit interval.

Journaling stores raw operations that allow MongoDB to reconstruct the following:

- document insertion/updates.
- index modifications.
- metadata changes to the namespace files.
- creation and dropping of databases and their associated data files.

As *write operations* occur, MongoDB writes the data to the **private view** in RAM and then copies the write operations in batches to the journal. The journal stores the operations on disk to ensure durability. Each journal entry describes the bytes the write operation changed in the data files.

MongoDB next applies the journal's write operations to the **shared view**. At this point, the **shared view** becomes inconsistent with the data files.

At default intervals of 60 seconds, MongoDB asks the operating system to flush the **shared view** to disk. This brings the data files up-to-date with the latest write operations. The operating system may choose to flush the **shared view** to disk at a higher frequency than 60 seconds, particularly if the system is low on free memory.

When MongoDB flushes write operations to the data files, MongoDB notes which journal writes have been flushed. Once a journal file contains only flushed writes, it is no longer needed for recovery, and MongoDB either deletes it or recycles it for a new journal file.

As part of journaling, MongoDB routinely asks the operating system to remap the **shared view** to the **private view**, in order to save physical RAM. Upon a new remapping, the operating system knows that physical memory pages can be shared between the **shared view** and the **private view** mappings.

The interaction between the **shared view** and the on-disk data files is similar to how MongoDB works *without* journaling, in which MongoDB asks the operating system to flush in-memory changes back to the data files every 60 seconds.

10.2 Couchbase Server

The storage engine used in Couchbase Server 3.0 is called Couchstore. It is written in C++ and has an append-only B+Tree design similar to one used in Apache CouchDB and Couchbase Server 2.0 (Erlang implementation).

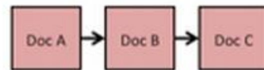
In general, B+Trees are well-known to be used in filesystems and database storage structures, and more commonly used than simpler heap files or other linked-list based data structures.

In Couchstore implementation both data and index files are opened in append-only mode. It is impossible to corrupt data or index files, as updates go to the end of the file. No in-place updates occur and thus files are never could in an inconsistent state. Besides being extremely robust, append-only approach leads to effective writers that do not disturb readers at all, allowing concurrent access without read locks.

The file header containing B+Tree root and internal nodes must be re-appended each time the tree structure has been changed. The data file grows without any bounds as it is modified containing more and more stale data. To work around this, files needs to be periodically compacted by collecting and writing live data to a new file, and replacing the old file with the new one after then. Compaction scans through the current data and index files and creates new data and index files, without the outdated and deleted items marked for cleanup. During compaction, the B+Trees are balanced and the reduce values are re-computed for the new tree.

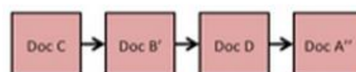
Finally, to catch-up with the active workload that might have changed the old partition data file during compaction, Couchbase copies over the data that was appended since the start of the compaction process to the new partition data file, so that it is up-to date. The new index file is also updated in this manner. The old partition data and index file are then deleted and the new data and index files are used.

Initial file layout:



Update some data:

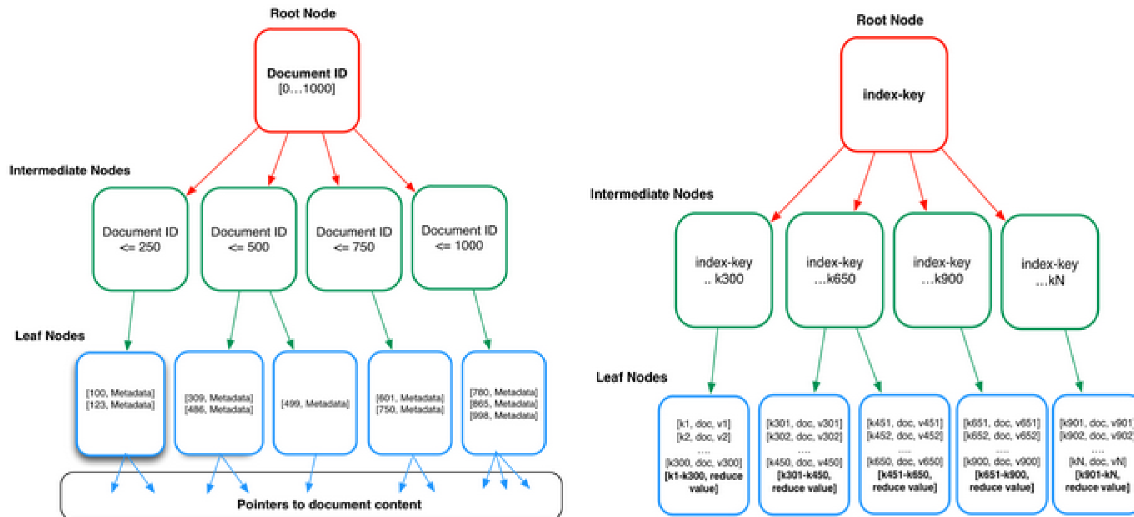
After compaction:



The compaction process runs in background and the switching between the old file and the new file happens without any database locking. The process can be triggered automatically when the percentage of the stale data inside the files grows more than a certain configured value, or manually. As the compaction process requires certain amount of compute and IO resources, it can be scheduled for a particular time of day.

Now we will go a bit deeper into the used file format and the B+Tree implementation details. Documents in Couchbase Server are stored in Buckets, as it was described earlier. Each Bucket is partitioned into vBuckets (1,024 by default). Each Partition is served in the main memory separately. And then multiple files are used for on-disk storage - a data file per vBucket, multiple index-files (active, replica, temp) per design document (an abstraction for JS Map-Reduce in Couchbase), and one master file that has metadata related to design documents and view definitions.

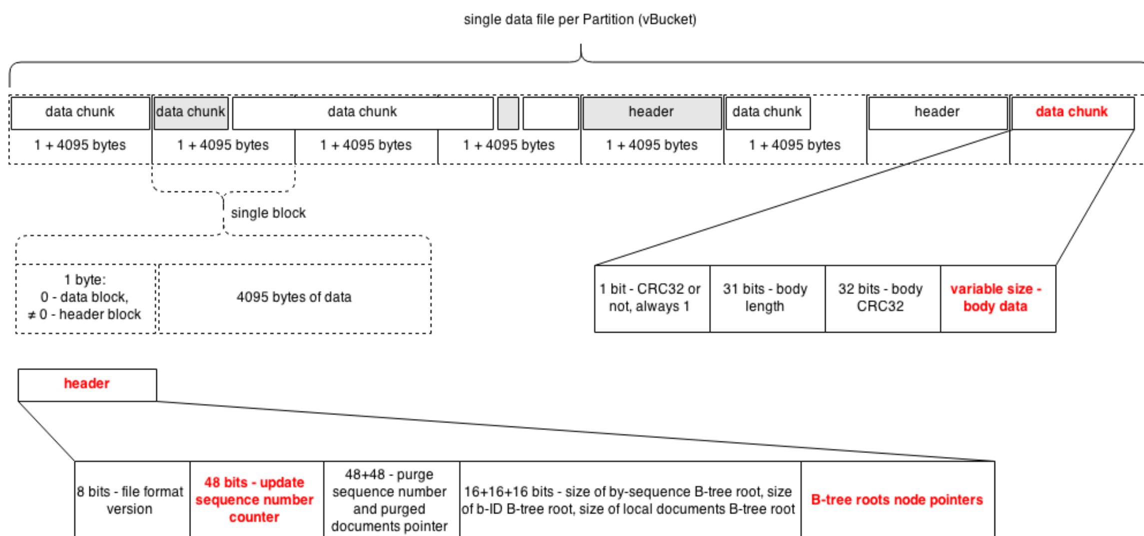
Data and index files in Couchbase Server are organized as B+Trees. The root nodes (shown in red) contains pointers to the intermediate nodes, which contain pointers to the leaf nodes (shown in blue). In the case of data files, the root and intermediate nodes track the sizes of documents and other useful information about their sub-trees.



The leaf nodes store the document id, document metadata and pointers to the document content. For index files, the root and intermediate nodes track the outputted map-function result and the reduce values under their sub-trees.

By examining the size of the documents tracked by the root node in the B+Tree file structure, the ratio of the actual size to the current size of the file is calculated. If this ratio hits a certain threshold that is configurable, an online compaction process is triggered or delayed until scheduled time as was mentioned above.

Below the logical B+Tree organization the reads and writes are performed on 4096-bytes blocks. The first byte of every block is reserved and identifies whether it's a data block (zero) or a header block (nonzero).



In case a file header needs to be located the file is scanned backwards: seek to the last block boundary, read one byte, and keep skipping back 4,096 bytes until the byte read is nonzero. A file

may contain many headers as it grows, since a new updated one is appended whenever data mutated. The current header is the last one in the file. File may contain stale headers and data chunks like shown in gray on the picture above.

The data in the file is grouped into variable-length chunks (above the block level, i.e. one data chunk can occupy several blocks) containing body length, CRC32 checksum of body and body data.

File header is crucial for locating the data inside the file. It contains update sequence number for the handled Partition and B+Trees information like their sizes and root node pointers.

The data file contains two important B+Trees inside:

- The by-ID index, which maps document ID to its value position within the file;
- The by-sequence index, which maps sequence numbers (database change numbers, which increase monotonically with every change) to document IDs and values;

The by-ID index is used for a document lookup by its ID. Each intermediate node, as pictured above, contains a key range. To get a document by its ID, algorithm starts from the file header, gets to the root B+Tree node of the by-ID index, and then traverse to the leaf node that contains the pointer to the actual document position in the file, finally reading the document data. When data mutation occurs, the whole traversal path needs to be updated and re-appended to the end of the file.

The by-sequence index is used internally in replication, compaction and other processes for operations like getting documents in their chronological update order, locating all the documents since a particular sequence number (for replicating only the recent updates) etc.

We will not cover here the Views or underlying index files details that have very similar storage principles for Couchbase Server 3.0, but going to be completely rethought in Couchbase Server 4.0 with the ForestDB data storage engine.

11. About the Authors

Vladimir Starostenkov is a Senior R&D Engineer at Altoros. He is focused on implementing complex software architectures, including data-intensive systems and Hadoop-driven apps. Having background in computer science, Vladimir is passionate about artificial intelligence and machine learning algorithms. His NoSQL and Hadoop studies were published in NetworkWorld, CIO.com, and other industry media.

***Altoros** brings Cloud Foundry-based "software factories" and NoSQL-driven "data lakes" into organizations through training, deployment, and integration. With 250+ employees across 9 countries in Europe and Americas, Altoros is the company behind some of the world's largest Cloud Foundry and NoSQL deployments. For more, please visit www.altoros.com.*

To download more NoSQL guides and tutorials:

- *check out our resources page,*
- *subscribe to the blog,*
- *or follow @altoros for daily updates.*