Couchbase

Couchbase Server Under the Hood

An Architectural Overview

Couchbase Server is an open-source distributed NoSQL document-oriented database for interactive applications, uniquely suited for those needing a flexible data model, easy scalability, consistent high-performance, and always-on 24x365 characteristics. This paper dives deep into the internal architecture of Couchbase Server. It assumes you have a basic understanding of Couchbase Server and are looking for greater technical understanding of what's under the hood.

High-Level Deployment Architecture

Couchbase Server has a true shared-nothing architecture. It is setup as a cluster of multiple servers behind an application server. Figure 1 shows two application servers interacting with a Couchbase cluster. It illustrates that documents stored in Couchbase can be replicated to other servers in a cluster. Replication will be discussed later.

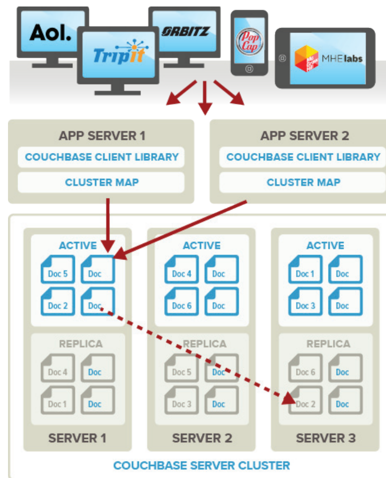

Figure 1: Couchbase Server deployment architecture

You can store JSON documents or binary data in Couchbase Server. Documents are uniformly distributed across the cluster and stored in data containers called **buckets**. Buckets provide a logical grouping of physical resources within a cluster. For example, when you create a bucket in Couchbase, you can allocate a certain amount of RAM for caching data and configure the number of replicas.

By design, each bucket is split into 1024 logical partitions called vBuckets. **Partitions** are mapped to servers across the cluster, and this mapping is stored in a lookup structure called the **cluster map**. Applications can use Couchbase's smart clients to interact with the server. Smart clients hash the **document ID**, which is specified by the application for every document added to Couchbase. As shown in Figure 2, the client applies a hash function (CRC32) to every document that needs to be stored in Couchbase, and the document is sent to the server where it should reside.

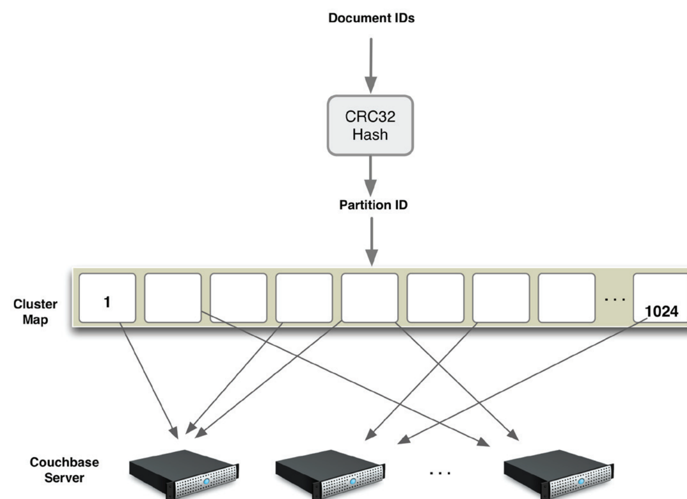

Figure 2: Mapping document IDs to partitions and then to servers

You can replicate a bucket up to three times within a cluster. Because Couchbase Server splits buckets into partitions, a server manages only a subset of active and replica partitions. This means that at any point in time, for a given partition, only one copy is active with zero or more replica partitions on other servers. If a server hosting an active partition fails, the cluster promotes one of the replica partitions to active, ensuring the application can continue to access data without any downtime.

Client Architecture

To talk to Couchbase Server, applications use memcached compatible smart-client SDKs in a variety of languages including Java, .NET, PHP, Python, Ruby, C, and C++. These clients are cluster topology aware and get updates on the cluster map. They automatically send request from the application to the appropriate server.

So how does data flow from a client application to Couchbase Server? And what happens within a Couchbase server when it receives a request to read and write data?

After a client first connects to the cluster, it requests the cluster map from the Couchbase cluster and maintains an open connection with the server for streaming updates. The cluster map is shared with all the servers in a Couchbase cluster and with the Couchbase clients. As shown in the Figure 3, data flows from a client to the server using the following steps:

1. An application interacts with an application resulting in the need to update or retrieve a document in Couchbase Server.

2. The application server contacts Couchbase Server via the smart client SDKs.

3. The smart client SDK takes the document that needs to be updated and hashes its document ID to a partition ID. With the partition ID and the cluster map, the client can figure out on which server and on which partition this document belongs. The client can then update the document on this server.

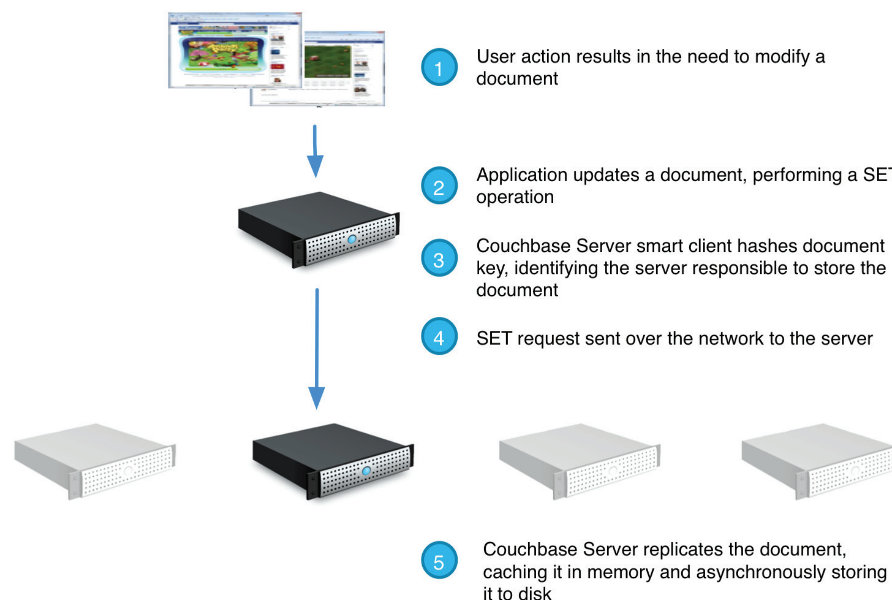

Figure 3: Flow of data from an application to a Couchbase Server cluster

4. (and 5.) When a document arrives in a cluster, Couchbase Server replicates the document, caches it in memory and asynchronously stores it on disk. More details on this process in the next paragraph.

In Couchbase Server, mutations happen at a document level. Clients retrieve the entire document from the server, modify certain fields, and write the entire document back to Couchbase. When Couchbase receives a request to write a document as shown in Figure 4 below, the following occurs:

1. Every server in a Couchbase cluster has its own object-managed cache. The client writes a document into the cache, and the server sends the client a confirmation. By default, the client does not have to wait for the server to persist and replicate the document as it happens asynchronously. However, using the client SDKs, you can design your application to wait.

2. The document is added into the intra-cluster replication queue to be replicated to other servers within the cluster.

3. The document is also added into the disk write queue to be asynchronously persisted to disk. The document is persisted to disk after the disk-write queue is flushed.

4. After the document is persisted to disk, it's replicated to other Couchbase Server clusters using cross datacenter replication (XDCR) and eventually indexed.

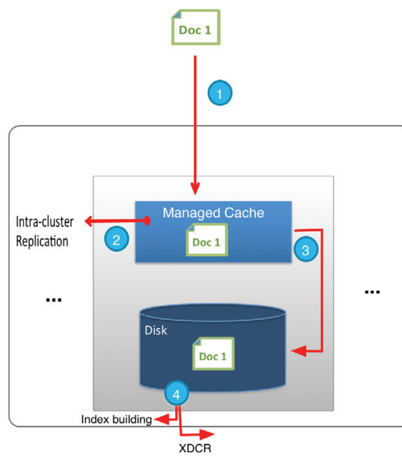

Figure 4: Data flow within Couchbase during a write operation

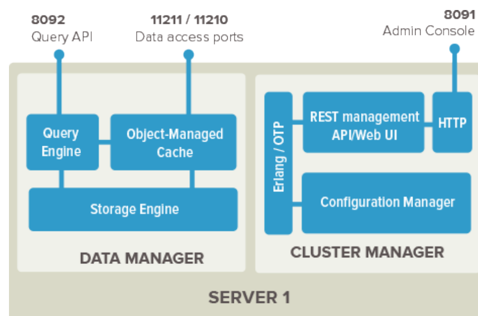

Figure 5: Node architecture diagram of Couchbase Server

Data Manager

In Couchbase Server, the data manager stores and retrieves data in response to data operation requests from applications. It exposes two ports to the network: one for non-smart clients (11211), that can be proxied via Moxi (a memcached proxy for couchbase) if needed, and the other for smart clients (11210). The majority of code in the data manager is written in C and C++.

The data manager has three parts - the object-managed cache, the storage engine, and the query engine. This section will explain the first two parts.

Object Managed Cache

Every server in a Couchbase cluster includes a built-in multi-threaded **object-managed cache**, which provides consistent low-latency for read and write operations. Your application can access it using memcached compatible APIs such as get, set, delete, append, and prepend.

As shown in the Figure 6, hashtables represent partitions in Couchbase. Each document in a partition will have a corresponding entry in the hashtable, which stores the document's ID, the metadata of the document, and potentially, the document's content. In other words, the hashtable contains metadata for all the documents and may also serve as a cache for the document content. Since the hashtable is in memory and look-ups are fast, it offers a quick way of detecting whether the document exists in memory or not. Read and write operations first go to the in-memory object-managed cache: if a document being read is not in the cache, it is fetched from disk. The server also tracks read and write operations in the last 24-hours in an access log file. If you need to restart a server, the access log file is used for server warmup, which we describe next.

When a Couchbase cluster is started for the first time, it creates necessary database files and begins serving data immediately. However, if there is already data on disk (likely because a node rebooted or due to a service restart), the servers need to read data off of disk to fill-up the cache. This process is called **warmup**.

Warmup consists of the following steps:

1. For each partition, Couchbase loads the keys and metadata for all the documents into memory.
2. If there is still space left in memory, Couchbase uses the access log file to prioritize which document's content should be loaded.

Warmup is a necessary step before Couchbase to serve requests and it can be monitored.

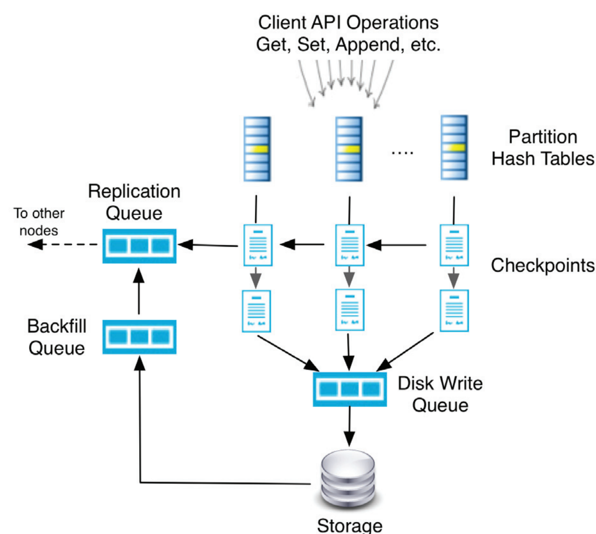

Figure 6: Persisting and replicating Couchbase checkpoints

Each hashtable has multiple worker threads that process read and write requests. Before a worker thread accesses the hashtable, it needs to get a lock. Couchbase uses fine-grained locking to boost request throughput; each hashtable has multiple locks, and each lock controls access to certain sections of that hashtable. As you add more documents to your cluster, the hashtable grows dynamically at runtime.

To keep track of document changes, Couchbase Server uses data structures called **checkpoints**. These are linked lists that keep track of outstanding changes that still need to be persisted by the storage engine and replicated to other replica partitions.

Documents are replicated to other servers by the **TAP replicator**. When a TAP connection is initiated on the source node, the hashtable for the particular partition is scanned. In parallel, a **backfill** process starts up and decides whether to bulk load data from disk. Since Couchbase enables you to have more data in the system than RAM available, there can be a significant amount of data that needs to be read from disk in order to transfer to another node. To handle this, the backfill process looks at the **resident item ratio**, which is amount of data cached in RAM versus total data in the system. If that ratio is lower than 90 percent, it bulk loads the whole partition from disk into a temporary RAM buffer. If the resident item ratio is higher than 90 percent, this process doesn't happen.

As the hashtable gets scanned, Couchbase begins transmitting the keys and documents over the TAP connection. Anything that is cached in RAM already is sent very quickly, anything else is either taken from that temporary buffer space (if available) or read in a one-off fashion from disk. During this period, mutations can change documents in memory. These changes are tracked using **checkpoints**. After the documents read by the backfill process have been replicated, the TAP replicator picks up changes from the checkpoint and replicates them. Working set metadata is also replicated in this process and maintained on the replica servers to reduce latencies after a rebalance or failover operation.

Once the server copies and synchronizes all data, it performs a **switchover**. Technically it could be possible for one server to transmit changes so quickly into a partition that the second server could never catch up and switch over. In practice, this never happens. Any brief delay on the client side between requests is enough to get the last bits over to the other server and it would be extremely rare for internode communication to be drastically slower than client-server communication.

Disk writes in Couchbase Server are asynchronous. This means that as soon as the document is placed into the hashtable, and the mutation is added to the corresponding checkpoint, a success response is sent back to the client. If a mutation already exists for the same document, deduplication takes place, and only the latest updated document is persisted to disk. In Couchbase 2.1, there are multiple worker threads so that multiple processes can simultaneously read and write data from disk to maximize disk I/O throughput. These threads are assigned exclusively to certain groups of partitions. This ensures that only those assigned worker threads can serve the partition at a given time.

By default, there are three worker threads per data bucket, with two reader workers and one writer worker for the bucket. But this number can be increased to a max total of 8 worker threads.

Couchbase Server processes read operations by looking up the document ID from the corresponding active partitions hashtable. If the document is in the cache (your application's working set), the document body is read from the cache and returned to the client. If the document is not found in the cache, it is read from disk into the cache, and then sent to the client.

Disk fetch requests in Couchbase Server are batched. Documents are read from the underlying storage engine, and corresponding entries in the hashtable are filled. If there is no more space in memory, Couchbase needs to free up space from the object-managed cache. This process is called **ejection**. Couchbase tracks documents that are not recently accessed using the not-recently-used (**NRU**) metadata bit. Documents with the NRU bit set are ejected out of cache to the disk. Finally, after the disk fetch is complete, pending client connections are notified of the asynchronous I/O completion so that they can complete the read operation.

To keep memory usage per bucket under control, Couchbase Server periodically runs a background task called the **item pager**. The high and low watermark limits for the item pager is determined by the bucket memory quota. The default high watermark value is 75 percent of the bucket memory quota, and the low watermark is 60 percent of the bucket memory quota. These watermarks are also configurable at runtime. If the high watermark is reached, the item pager scans the hashtable, ejecting items that are not recently used.

Couchbase Server supports document expiration using time to live (TTL). By default, all documents have a **TTL** of 0, which means the document will be kept indefinitely. However, when you add, set, or replace a document, you can specify a custom TTL. Expiration of documents is lazy. The server doesn't immediately delete items from disk. Instead, it runs a background task every 60 minutes (by default) called the **expiry pager**. The expiry page tracks and removes expired documents from both the hashtable as well as the storage engine through a delete operation.

Storage Engine

With Couchbase's append-only storage engine design, document mutations go only to the end of the file. If you add a document in Couchbase, it gets added at the end of the file. If you append a document in Couchbase, it gets recorded at the end of the file. Files are open always in append mode, and there are never any in-place updates. This improves disk write performance as updates are written sequentially. Figure 7 illustrates how Couchbase Server stores documents:

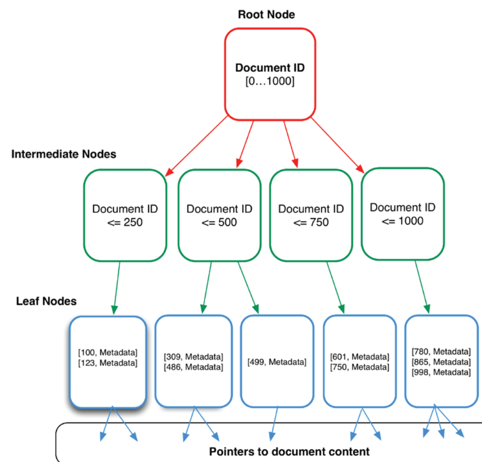

Figure 7: Data file b-tree structure in Couchbase Server

Couchbase Server uses multiple files for storage. It has a data file per partition, which we call **data file**, and multiple **index files**, which are either active or replicated indexes. The third type of file is a **master file**, which has metadata related to Couchbase views, and will be discussed later in the section on views.

As shown in Figure 7, Couchbase Server organizes data files as b-trees. The root nodes shown in red contain pointers to intermediate nodes. These intermediate nodes contain pointers to the leaf nodes shown in blue. The root and intermediate nodes also track the sizes of documents under their sub-tree. The leaf nodes store the document ID, document metadata and pointers to document content. For index files the root and intermediate nodes track index items under their sub-tree.

To prevent the data file from growing too large and eventually eating up all your disk space, Couchbase periodically cleans up stale data from the append-only storage. It calculates the ratio of the actual file size to the current file. If this ratio goes below a certain threshold (configurable through the admin UI), a process called **compaction** is triggered. Compaction scans through the current data file and copies non-stale data into a new data file. Since it is an online operation, Couchbase can still service requests. When the compaction process is completed, Couchbase copies over the data modified since the start of the compaction process to the new data file. The old data file is then deleted and the new data file is used until the next compaction cycle.

Cluster Manager

The cluster manager supervises server configuration and interaction between servers within a Couchbase cluster. It is a critical component that manages replication and rebalancing operations in Couchbase. Although the cluster manager executes locally on each cluster node, it elects a clusterwide **orchestrator node** to oversee cluster conditions and carry out appropriate cluster management functions.

If a machine in the cluster crashes or becomes unavailable, the cluster orchestrator notifies all other machines in the cluster, and promotes to active status all the replica partitions associated with the server that's down. The cluster map is updated on all the cluster nodes and the clients. This process of activating the replicas is known as **failover**. You can configure failover to be automatic or manual. Additionally, you can trigger by an external monitoring script via the REST API.

If the orchestrator node crashes, existing nodes will detect that it is no longer available and will elect a new orchestrator immediately so that the cluster continues to operate without disruption.

At a high level, there are three primary cluster manager components on each Couchbase node:

- **The heartbeat watchdog** periodically communicates with the cluster orchestrator using the **heartbeat protocol.** It provides regular health updates of the server. If the orchestrator crashes, existing cluster server nodes will detect the failed orchestrator and elect a new orchestrator.

- **The process monitor** monitors what the local data manager does, restarts failed processes as required, and contributes status information to the heartbeat process.

- **The configuration manager** receives, processes, and monitors a node's local configuration. It controls the cluster map and active replication streams. When the cluster starts, the configuration manager pulls configuration of other cluster nodes and updates its local copy.

Along with handling metric aggregation and consensus functions for the cluster, the cluster manager also provides a REST API for cluster management and a rich web admin UI:

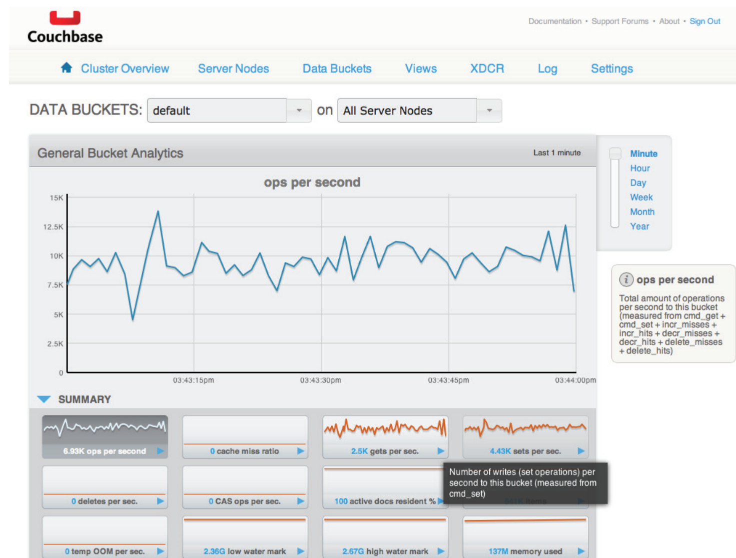

Figure 8: Couchbase Server admin UI

As shown in the Figure 8, the admin UI provides a centralized view of the cluster metrics across the cluster. You can drill down on the metrics to get an idea of how well a particular server is functioning or if there are any areas that need attention. The cluster manager is built on Erlang/OTP, a proven environment for building and operating fault-tolerant distributed systems.

In addition to per-node operations that are always executing at each node in a Couchbase Server cluster, certain components are active on only one node at a time, including:

- **The rebalance orchestrator**. When the number of servers in the cluster changes due to scaling out or failures, data partitions must be redistributed. This ensures that data is evenly distributed throughout the cluster, and that application access to the data is load balanced evenly across all the servers. This process is called **rebalancing**. You can trigger rebalancing using an explicit action from the admin UI or through a REST call.

  When a rebalance begins, the rebalance orchestrator calculates a new cluster map based on the current pending set of servers to be added and removed from the cluster. It streams the cluster map to all the servers in the cluster. During rebalance, the cluster moves data via partition migration directly between two server nodes in the cluster. As the cluster moves each partition from one location to another, an atomic and consistent switchover takes place between the two nodes, and the cluster updates each connected client library with a current cluster map. Throughout migration and redistribution of partitions among servers, any given partition on a server will be in one of the following states:

  - **Active**. The server hosting the partition is servicing all requests for this partition.
  - **Replica**. The server hosting the partition cannot handle client requests, but it will receive replication commands. Rebalance marks destination partitions as replica until they are ready to be switched to active.
  - **Dead**. The server is not in any way responsible for this partition.

- **The node health monitor** receives heartbeat updates from individual nodes in the cluster, updating configuration and raising alerts as required.
- **The partition state and replication manager** is responsible for establishing and monitoring the current network of replication streams.

Query Engine

With Couchbase Server, you can easily index and query JSON documents. Secondary indexes are defined using **design documents** and **views**. Each design document can have multiple views, and each Couchbase bucket can have multiple design documents. Design documents allow you to group views together. Views within a design document are processed sequentially and different design documents can be processed in parallel at different times.

Views in Couchbase are defined in JavaScript using a **map** function, which pulls out data from your documents and an optional **reduce** function that aggregates the data emitted by the map function. The map function defines what attributes to build the index on.

Indexes in Couchbase are distributed; each server indexes the data it contains. As shown in Figure 9, indexes are created on active documents and optionally on replica documents.

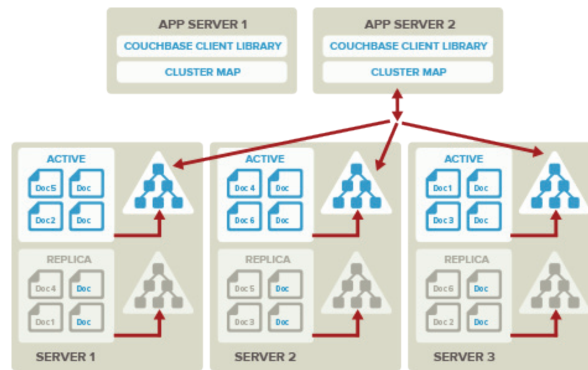

Figure 9: Distributed indexing and querying in Couchbase Server

Applications can query Couchbase Server using any of the Couchbase SDKs, even during a rebalance. Queries are sent via port 8092 to a randomly selected server within the cluster. The server that receives the query sends this request to other servers in the cluster, and aggregates results from them. The final query result is then returned to the client.

During initial view building, on every server, Couchbase reads the partition data files and applies the map function across every document. Before writing the index file, the reduce function is applied to the b-tree nodes - if the node is a non-root node, the output of the reduce function is a partial reduce value of its sub-tree. If the node is a root node, the reduce value is a full reduce value over the entire b-tree.

As shown in Figure 10, the map reduce function is applied to every document, and output key value pairs are stored in the index.

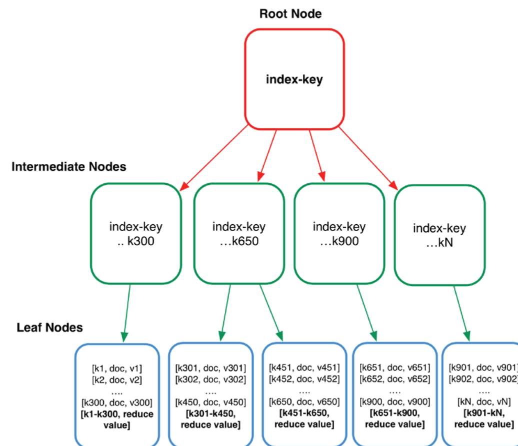

Figure 10: Index file b-tree structure in Couchbase Server

Like primary indexes, you can emit one or more attributes of the document as the key and query for an exact match or a range. For example, as shown in Figure 11, a secondary index is defined using the name attribute.

VIEW CODE

Map

```
function (doc, meta) {
  if(doc.name)
  emit(doc.name, null);
}
```

Index definition

Key

Value

Figure 11: Defining a secondary index with attribute name

Views in Couchbase are built asynchronously and hence **eventually indexed**. By default, queries on these views are eventually consistent with respect to the document updates. Indexes are created by Couchbase Server based on the view definition, but updating of these indexes can be controlled at the point of data querying, rather than each time data is inserted. The **stale** query parameter can be used to control when an index gets updated.

Indexing in Couchbase Server is incremental. If a view is accessed for the first time, the map and reduce functions are applied to all the documents within the bucket. Each new access to the view processes only the documents that have been added, updated, or deleted since the last time the view index was updated. For each changed document, Couchbase invokes the corresponding map function and stores the map output in the index. If a reduce function is defined, the reduce result is also calculated and stored. Couchbase maintains a **back index** that maps IDs of documents to a list of keys the view has produced. If a document is deleted or updated, the key emitted by the map output is removed from the view and the back index. Additionally, if a document is updated, new keys emitted by the map output are added to the back index and the view.

Because of the incremental nature of the view update process, information is only appended to the index stored on disk. This helps ensure that the index is updated efficiently. Compaction will optimize the index size on disk and the structure.

During a rebalance operation, partitions change state – the original partition transitions from active to dead, and the new one transitions from replica to active. Since the view index on each node is synchronized with the partitions on the same server, when the partition has migrated to a different server, the documents that belong to a migrated partition should not be used in the view result anymore. To keep track of the relevant partitions that can be used in the view index, Couchbase Server has a modified indexing data structure called the **B-Superstar index**.

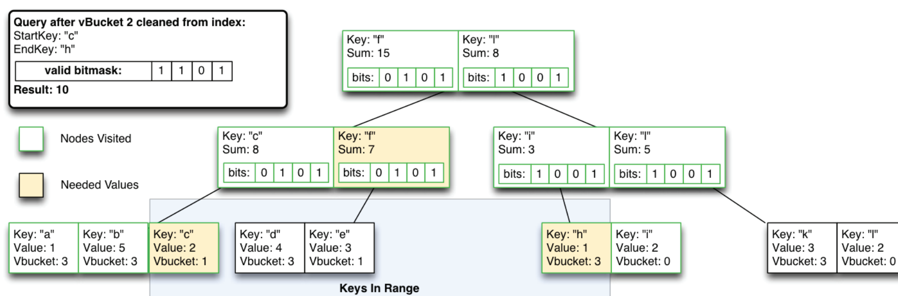

Figure 12: A distributed range query touches only a subset of partitions

In the B-Superstar index, each non-leaf B-tree node has a bitmask with 1024 bits – each bit corresponding to a partition. As shown in Figure 12 at query time, when a query engine traverses a b-tree, if the partition id of a key-value pair is not set in the bitmask, it is not factored into the query result. To maintain consistency for indexes during rebalance or failover, partitions can be excluded by a single bit change in the bitmask. The server updates the bitmask every time the cluster manager tells the query engine that partitions changed to active or dead states.

Cross Datacenter Replication

Cross datacenter replication provides an easy way to replicate active data to multiple, geographically diverse datacenters either for disaster recovery or to bring data closer to its users.

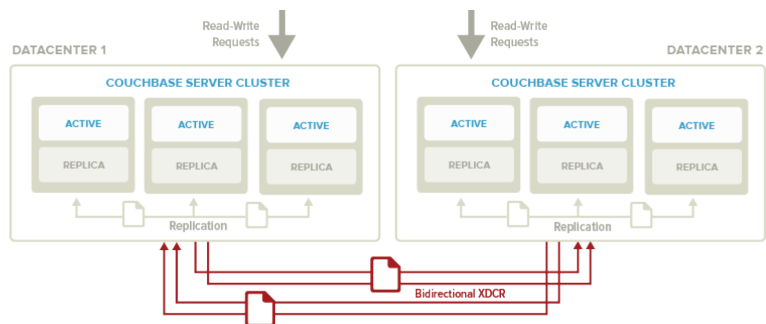

Figure 13: Cross datacenter replication architecture in Couchbase Server

In Figure 13 you can see that XDCR and intra-cluster replication occurs simultaneously. Intra-cluster replication is taking place within the clusters at both Datacenter 1 and Datacenter 2, while at the same time XDCR is replicating documents across datacenters. On each node, after a document is persisted to disk, XDCR pushes the replica documents to other clusters. On the destination cluster, replica documents received will be stored in the Couchbase object managed cache so that replica data on the destination cluster can undergo low latency read/write operations.

XDCR can be setup on a per bucket basis. Depending on your application requirements, you might want to replicate only a subset of the data in Couchbase Server between two clusters. With XDCR you can selectively pick which buckets to replicate between two clusters in a unidirectional or bidirectional fashion. As shown in Figure 14, bidirectional XDCR is setup between Bucket C (Cluster 1) and Bucket C (Cluster 2). There is unidirectional XDCR between Bucket B (Cluster 1) and Bucket B (Cluster 2). Bucket A is not replicated.

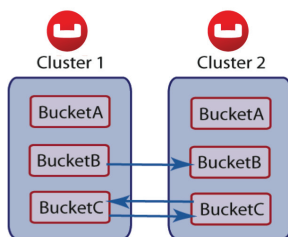

Figure 14: Replicating selective buckets between two Couchbase Server clusters

XDCR supports **continuous replication** of data. Data mutations are replicated to the destination cluster after they are written to disk. By default, there are 32 data streams per server per XDCR connection. These streams are spread across the partitions. They move data in parallel from the source cluster to the destination cluster.

XDCR is **cluster topology aware**. The source and destination clusters can have different number of servers. If a server in the destination cluster goes down, XDCR is able to get the updated cluster topology information and continue replicating data to the available servers in the destination cluster.

XDCR is **push-based**. The source cluster regularly checkpoints the XDCR replication queue per partition and keeps track of what data the destination cluster last received. If the replication process is interrupted for example due to a server crash or intermittent network connection failure, it is not required to restart replication from the beginning. Instead, once the replication link is restored, replication can continue from the last checkpoint seen by the destination cluster.

By default, XDCR in Couchbase is **designed to optimize bandwidth**. This includes optimizations like mutation de-duplication as well as checking the destination cluster to see if a mutation is required to be shipped across. If the destination cluster has already seen the latest version of the data mutation, the source cluster does not send it across. However, in some use cases with active-active XDCR, you may want to skip checking whether the destination cluster has already seen the latest version of the data mutation, and optimistically replicate all mutations. Couchbase 2.1 supports optimistic XDCR replication. For documents with size within the XDCR size threshold (which can be configured), no pre-check is performed and the mutation is replicated to the destination cluster.

Within a cluster, Couchbase Server provides strong consistency at the document level. On the other hand, XDCR also **provides eventual consistency across clusters**. Built-in conflict resolution will pick the same "winner" on both the clusters if the same document was mutated on both the clusters before it was replicated across. If a conflict occurs, the document with the most updates will be considered the "winner." If both the source and the destination clusters have the same number of updates for a document, additional metadata such as numerical sequence, CAS value, document flags and expiration TTL value are used to pick the "winner." XDCR applies the same rule across clusters to make sure document consistency is maintained.

Figure 15 shows an example of how conflict detection and resolution works in Couchbase Server. As you can see, bidirectional replication is set up between Datacenter 1 and Datacenter 2 and both the clusters start off with the same JSON document (Doc 1). In addition, two additional updates to Doc 1 happen on Datacenter 2. In the case of a conflict, Doc 1 on Datacenter 2 is chosen as the winner because it has seen more updates.

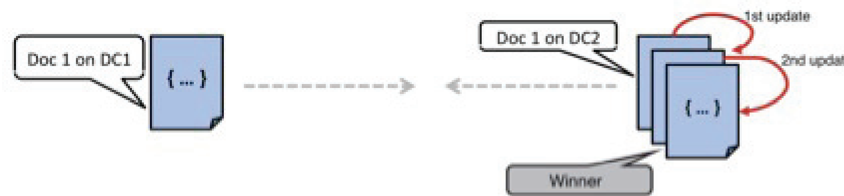

Figure 15: Conflict detection strategy in Couchbase Server

Conclusion

This paper describes the architecture of Couchbase Server but the best way to get to know the technology is to download and use it. Couchbase Server is a good fit for a number of use cases including social gaming, ad targeting, content store, high availability caching, and more. In addition Couchbase also has a rich ecosystem of adapters such as a Hadoop connector and a plug-in for ElasticSearch. To download Couchbase Server, visit http://www.couchbase.com/downloads.

About Couchbase

We're the company behind the Couchbase open source project, a vibrant community of developers and users of Couchbase document-oriented database technology. Our flagship product, Couchbase Server, is a packaged version of Couchbase technology that's available in Community and Enterprise Editions. We're known for our easy scalability, consistent high performance, 24x365 availability, and a flexible data model. Companies like AOL, Cisco, Concur, LinkedIn, Orbitz, Salesforce.com, Shuffle Master, Zynga and hundreds of others around the world use Couchbase Server for their interactive web and mobile applications. www.couchbase.com

© 2013 Couchbase all rights reserved | www.couchbase.com