
Couchbase
Couchbase Under the Hood
An Architectural Overview

Couchbase Under the Hood

An Architectural Overview

Table of Contents

INTRODUCTION

Couchbase is an open source, distributed, document-oriented, NoSQL database with a core architecture that supports a flexible JSON data model, easy scalability, consistent high performance, mobile synchronization, always-on 24x365 characteristics, and advanced security.

As a flexible document database, Couchbase supports multiple data access patterns on top of a flexible JSON data model. With its integrated features, Couchbase consolidates multiple layers into a single platform that would otherwise require separate solutions to work together. Couchbase is uniquely able to provide the performance of a caching layer, the flexibility of a source of truth, and the reliability of a system of record, eliminating the need to manage data models and consistency between multiple systems, learn different languages and APIs, and manage independent technologies.

This paper describes how the internal components of the Couchbase database (Server and Mobile) operate with one another. It assumes you have a basic understanding of Couchbase and are looking for greater technical understanding of how things operate beneath the surface.

Essential NoSQL requirements and features

NoSQL databases evolved from enterprise relational databases to address performance and delivery deficiencies. Relational databases tend to operate primarily as systems of record, maintaining transactional data in a highly consistent manner. But several architectural principles (normalization of objects, single node transactional design, two-phase commit) have made them difficult to scale to larger workloads while simultaneously delivering responsive and highly available applications.

Pragmatic business needs for more advanced technical requirements have pushed NoSQL to the forefront. These modern requirements have driven Couchbase's development from day one:

- **Agile development**
- **Scalable performance**
- **System manageability**

Couchbase has been focused on setting a high standard in each of these areas. The result is a robust and accessible set of features with solid performance and ease of management in a single NoSQL database. Learn more about how Couchbase delivers on these core database requirements in the following table and later in the text of this paper.

Develop with agility	Perform at any scale	Manage with ease
• Flexible JSON data model supports continuous delivery. Make schema changes without downtime. • Leverage common SQL query patterns for joins, aggregations, and more. • Extract value using a broad set of key capabilities (mobile sync, full-text search, real-time analytics, etc.). • No hassle scale-out.	• Memory- and network-centric architecture, with an integrated cache delivering high throughput and sub-millisecond latency. • Always-on, fault tolerant design. • Consistent performance at any scale. • Isolated and independent scaling of workloads, with no downtime or code changes.	• Global deployment with low write latency using active-active cross datacenter replication. • Infrastructure agnostic support across physical, virtual, cloud and containerized environments. • Microservices architecture with built-in auto-sharding, replication, and failover. • Full-stack security with end-to-end encryption and role-based access control.

These principles have been built into the very core of the database engine to ensure low latency and reliable, yet easy to manage, replication. Around this core is built a set of data access components that run and scale independently of each other. These are delivered through a unified programming API, established security capabilities, and external technology integrations.

Core design principles

To effectively deliver the above features, three guiding principles have been followed when developing Couchbase: memory and network-centric architecture, workload isolation, and an asynchronous approach to everything.

Memory and network-centric architecture

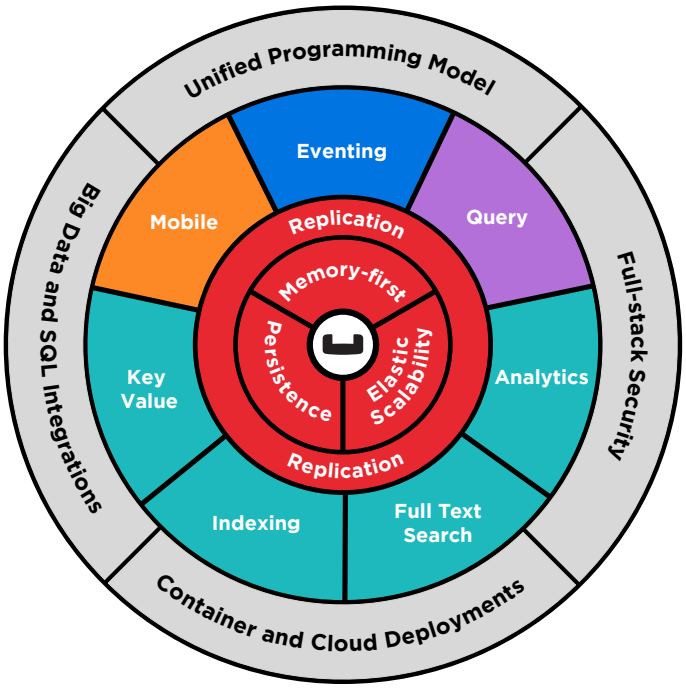
- The most used data and indexes are transparently maintained in memory for fast reads.
- Writes are performed in memory and replicated or persisted synchronously or asynchronously.
- Internal Database Change Protocol (DCP) streams data mutations from memory to memory at network speed to support replication, indexing and mobile synchronization.

Workload isolation

- All databases perform different tasks in support of an application. These include persisting, indexing, querying, and searching data. Each of these workloads has slightly different performance and resource requirements.
- Multi-Dimensional Scaling (MDS) isolates these workloads from one another at both a process and a node level.
- MDS allows these workloads to be scaled independently from one another and their resources to be optimized as necessary.
- Couchbase manages the topology, process management, statistics gathering, high availability and data movement between these services transparently.

Asynchronous approach to everything

- Traditional databases increase latency and block application operations while running synchronous operations, for example, persisting data to disk or maintaining indexes.
- Couchbase allows write operations to happen at memory and network speeds while asynchronously processing replication, persistence and index management.
- Spikes in write operations don't block read or query operations, while background processes will persist data as fast as possible without slowing down the rest of the system.
- Durability and consistency options are available allowing the developer to decide when and where to increase latency in exchange for durability and consistency.



Couchbase NoSQL Database

JSON DATA MODEL AND ACCESS METHODS

This section outlines the foundational JSON data model handling in Couchbase, then introduces the multiple ways to access that data. These methods include basic key-value operations, SQL querying, full-text searching, real-time analytics, server-side eventing, and mobile application synchronization.

JSON data model

The JSON data model supports basic and complex data types: numbers, strings, nested objects, and arrays. JSON provides rapid serialization and deserialization, is native to JavaScript, and is the most common REST API data format. Consequently, JSON is extremely convenient for web application programming.

Couchbase stores data as individual documents, comprised of a key and a value. When the value is JSON-formatted, Couchbase provides rich access capabilities; when not, the document is stored as a binary BLOB and has more limited access characteristics.

A document often represents a single instance of an application object (or nested objects). It can also be considered analogous to a row in a relational table, with the document attributes acting similar to a column. Couchbase provides greater flexibility than relational databases, allowing JSON documents with varied schemas and nested structures. Developers may express many-to-many relationships without requiring a reference or junction table. Subcomponents of documents can be accessed and updated directly as well, and multiple document schemas can be aggregated together into a virtual table with a single query.

Flexible, dynamic schema

In the document model, a schema is the result of an application's structuring of its documents: schemas are defined and managed by applications. A document's structure consists of its inner arrangement of attribute-value pairs. This is in contrast to the relational model where the database manages the schema. For example, both of the following JSON document examples are valid data models that Couchbase can manage and query. How the documents are designed or updated over time is up to the application developer – normalized or denormalized or a hybrid depending on the needs and evolution of the application.

Normalized – 4 documents	Denormalized – 1 document

invoice1:
{
"BillTo": "Lynn Hess",
"InvoiceDate": "2018-01-15",
"InvoiceNum": "ABC123",
"ShipTo": "H. Trisler, 41 Oak Drive"
}
invoice1:item1:
{
"InvoiceId": "1",
"Price": "100",
"Product": "Brake Pad",
"Quantity": "24"
}
 	
invoice1:item2:
{
"InvoiceId": "1",
"Price": "10",
"Product": "Rotor",
"Quantity": "5"
}
 	
invoice1:item3:
{
"InvoiceId": "1",
"Price": "20",
"Product": "Tire",
"Quantity": "2"
}
	invoice1:
	{
	"BillTo": "Lynn Hess",
	"InvoiceDate": "2018-01-15",
	"InvoiceNum": "ABC123",
	"ShipTo": "H. Trisler, 41 Oak Drive",
	"Items": [
	{ "Price": "100", "Product": "Brake Pad", "Quantity": "24" },
	{ "Price": "10", "Product": "Rotor", "Quantity": "5" },
	{ "Price": "20", "Product": "Tire", "Quantity": "2" }
	]
	}

Couchbase does not enforce uniformity: document structures can vary, even across multiple documents where each contain a type attribute with a common value. This allows differences between objects to be represented efficiently. It also allows a schema to progressively evolve for an application, as required – properties and structures can be added to the document without other documents needing to be updated in the same way. This allows applications to change their behavior without having to overhaul all the source data or take applications offline to make a basic change.

Document access methods

Managing JSON data is at the core of Couchbase's document database capabilities, but there are several ways for applications to access the data. Each of these methods is described in further detail later in this paper, but the following provides a basic explanation and coding example of using it.

Access method	Description	Example
Key-value	An application provides a document ID (the key), Couchbase returns the associated JSON or binary object. The inverse occurs with a write or update request.	**Java:** `JsonDocument myAirline = bucket.get("airline_5209");`
Query and Analytics	SQL-based query syntax to interact with JSON data, similar to relational databases, returns matching JSON results. Comprehensive DML, DQL and DDL syntax supports nested data and non-uniform schema.	**Python:** `N1QLQuery( 'SELECT fname, lname, age FROM default WHERE age > $age', age=22 )`
Full-text search	Using text analyzers with tokenization and language awareness, a search is made for a variety of field and boolean matching functions. Search returns document IDs, relevance scoring and optional context data.	**Java:** `SearchQuery.term("sushi") .field("reviews.content") .fuzziness(1);`
Eventing	Custom JavaScript functions are executed within the database as data changes or based on timers. Support for accessing and updating data, writing out to a log or calling out to an external system.	**JavaScript:** `function OnUpdate(doc, meta) { log('document', doc); doc["ip_num_start"] = ip_trim_func(doc["ip_start"]); tgt[meta.id]=doc; }`

In addition to the above server functions, data can also be synchronized with mobile applications. Couchbase Mobile is the end-to-end stack, comprised of Server, Gateway and Lite. On a mobile device or embedded system, data is created, updated, searched, and queried whether online or offline. The data can then be synchronized with Couchbase Server and used by both mobile and server-based applications.

Couchbase Mobile	Couchbase Lite SDK provides common create, update, and delete tasks as well as query, full-text search, and triggers on the device.
	
	Sync Gateway keeps data updated with a Couchbase Server database and other devices
		Java:
		`MutableDocument mutableDoc = new MutableDocument()`
		`.setFloat("version", 2.0F)`
		`.setString("type", "SDK");`
		`database.save(mutableDoc);`
		Swift (iOS):
		`let newDoc = MutableDocument()`
		`.setString(2.0, forKey: "version")`
		`.setString("SDK", forKey: "type")`
		`try database.saveDocument(newDoc)`

Keys, values, and sub-documents

Keys and values are fundamental parts of JSON documents and have some limits that are important to understand.

Keys

Each value is identified by a unique key, or ID, defined by the user or application when the item is originally created. The key is immutable: once the item is saved, the key cannot be changed.

Each key must be a UTF-8 string with no spaces. Special characters, such as (, %, /, ", and _, are acceptable, but the key may be no longer than 250 bytes and must be unique within its bucket.

Values

The maximum size of a value is 20 MB. Each document consists of one or more attributes, each of which has its own value. An attribute's value can be a basic type, such as a number, string, or boolean; or a complex type, such as an embedded document or an array.

JSON documents can be parsed, indexed, and queried by other Couchbase services. A value can also be any form of binary, though it won't be parsed, indexed, or queried.

Sub-documents

A sub-document is an inner component of a JSON document. The sub-document API uses a path syntax to specify attributes and array positions to read/write. This makes it unnecessary to transfer entire documents over the network when only partial modifications are required.

Document key: *user200*

```
{
  "age": 21,
  "fav_drinks": {
    "soda": [ "fizzy", "lemon" ]
    }
  "addresses": [
    { "street": "pine" },
    { "street": "maple" }
  ]
}
```

Path example	Result
`age – a numeric value`	`21`
`fav_drinks.soda – an array of strings`	`fizzy, lemon`
`fav_drinks.soda[0] – first string in array`	`fizzy`
`addresses[1].street – string value in second part of array`	`maple`

Key Couchbase concepts

Buckets

Buckets hold JSON documents – they are a core concept for document organization in Couchbase. Applications connect to a specific bucket that pertains to their application scope. Memory quotas are managed on a per-bucket and per-service basis. Security roles are applied to users with various bucket-level constraints.

In addition to standard Couchbase buckets, there are two specialized bucket types useful for different use cases.

Ephemeral buckets do not persist data but allow highly consistent in-memory performance, without disk-based fluctuations. This delivers faster node rebalances and restarts.

Memcached buckets also do not persist data. It is a legacy bucket type designed to be used alongside other database platforms specifically for in-memory distributed caching. Memcached buckets lack most of the core benefits of Couchbase buckets, including compression.

vBuckets

vBuckets are shards or partitions of data (and replicas) that are automatically distributed across nodes. They are managed internally by Couchbase and not interacted with directly by the user. The Couchbase SDK automatically distributes data and workloads across vBuckets.

Nodes

Couchbase nodes are physical or virtual machines that host single instances of Couchbase Server. Multiple instances of Couchbase Server cannot be installed on a node.

Clusters

A cluster consists of one or more nodes running Couchbase Server. Nodes can be added or removed from a cluster. Replication of data occurs between nodes and cross datacenter replication occurs between different clusters that are geographically distributed.

Services

The core of Couchbase is the data service which feeds and supports all the other systems and data access methods. A service is an isolate set of processes dedicated to a particular tasks. For example, indexing, search, or querying are each managed as separate services. One or more services can be run on one or more nodes as needed.

COUCHBASE SERVICES

Couchbase implements the above data access methods through a set of dedicated services, with data at its center. Each service has their own resource quotas and, where applicable, related indexing and inter-node communication capabilities. This provides several very flexible methods to scale services when needed – not just scaling up to larger machines or scaling out to more nodes. Couchbase provides both options, as well as the ability to scale specific services differently than one another. Multi-dimensional scaling is covered in a later section but is the foundation of these workload isolation and scaling capabilities.

This is different than other platforms where a monolithic set of services are installed on every node in a cluster. Instead, Couchbase use a core data capability that then feeds all the other services. A shared-nothing architecture allows user control over workload isolation. Small-scale environments can share the same workloads across one or more

nodes, while higher scale and performance can be achieved with dedicated nodes to handle specific workloads – the ultimate in scale-out capability. The cluster can be scaled in or out and its service topology changed on demand with zero interruption or change to the application.

Applications communicate directly with each service through a common SDK that is aware of the topology of the cluster and how services are configured. Developers do not have to know anything about how the services and nodes are configured, the SDK gets all the information it needs from the cluster manager. The same application can be deployed against a cluster of any size or configuration without change in behavior. Having this knowledge built into the SDK results in reduced latency (direct application to database access) and complexity (no proxy or router components).

The core data service handles all the document mutations and streams them to all the related services described below. The remainder of this section walks through each service and describes how they work on a node, in a cluster, and with each other.

Later in this paper, the Distributed Foundation section will discuss inter-node connectivity, data flow, cluster topology, and data streaming.

Data service and KV engine

The data service is the foundation for storing data in Couchbase Server. It must run on at least one node of every cluster – it is responsible for caching, persisting and serving data to applications and other services within the cluster. The principal component of the data service architecture is the key-value management system known simply as KV engine.

KV engine is composed of a multi-threaded, append-only storage layer on disk with a tightly integrated managed object cache. The cache provides consistent low latency for individual document read and, write operations and streams documents to other services via DCP.

Each node running the data service has its own KV engine process and is responsible for persisting and caching a portion of the overall dataset (both active and replica).

Managed object cache

The managed object cache of each node hashes the document into an in-memory hash table based upon the document ID (key). The hash table stores the key, the value and some metadata associated with each document. Since the hash table is in memory and lookups are fast, it offers a quick way of detecting whether the document exists in memory or not.

The cache is both read-through and write-through: if a document being read is not in the cache, it is fetched from disk and write operations are written to disk after being first stored in memory.

Disk fetch requests are batched to the underlying storage engine, and corresponding entries in the hash table are filled. 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.

Document expiration

Documents may also be set to expire using a time to live (TTL) setting. By default, all documents have a TTL of 0, meaning the document will be kept indefinitely. When you add, set, or replace a document, you can specify a custom TTL, at which time the document becomes unavailable and is marked for deletion (tombstone) to be cleaned up later.

Memory management

To keep memory usage of the cache under control, Couchbase employs a background task called the item pager. This pager runs periodically (to clean up expired documents) as well as being triggered based on high and low watermarks of memory usage. This high water mark is based off of the memory quota for a given bucket and can be changed at runtime. When the high watermark is reached, the item pager scans the hash table, ejecting eligible (persisted) items that are not recently used (NRU). It repeats this process until memory usage falls below the low watermark.

Compression

End-to-end data document compression is available across all features of the database using the open source Snappy library. Client capabilities, and the compression mode configured for each bucket, determine how compression will run.

Data can optionally be compressed by the client (SDK) prior to writing into a bucket, within memory of the bucket and on disk. It is also compressed between nodes of the cluster and to remote clusters.

Compression modes

Data is always compressed on disk, but each client and bucket can control whether it is also compressed on the wire and/or in memory. The SDKs communicate whether they will be sending or requesting compressed documents, and the compression mode of the bucket determines what happens within the database. The modes are as follows:

Off – Documents are actively decompressed before storing in memory. Clients receive the documents uncompressed.

Passive – Documents are stored in memory both compressed and uncompressed in memory, depending on how the client has sent them. Clients receive compressed documents if they are able and uncompressed if they are not.

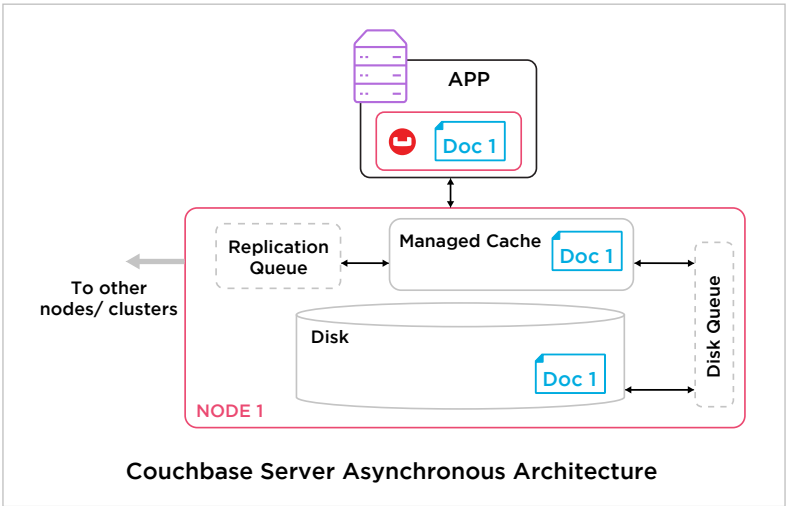
Active – Documents are actively compressed on the server, regardless of how they were received. Clients receive compressed data whenever it is supported by the client, even if it originally sent uncompressed data.

Compaction

Couchbase writes all data that you append, update and delete as files on disk. This can eventually lead to gaps in the data file, particularly when you delete data. You can reclaim the empty gaps in all data files by performing a process called compaction. For both data files and index files, perform frequent compaction of the files on disk to help reclaim disk space and reduce disk fragmentation. Auto-compaction is enabled by default for all buckets, but parameters can be adjusted for the entire cluster or for a specific bucket in a cluster.

Mutations

In Couchbase Server, mutations happen at a document level. Clients retrieve the entire document from the server, modify certain fields, and write the document updates back to Couchbase.



Couchbase Server Asynchronous Architecture

When Couchbase receives a request to write a document, the following occurs:

1. Every server in a Couchbase cluster has its own managed object 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.
2. The document is added into the intra-cluster replication queue to be replicated to other servers within the cluster.
3. The document is 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 clusters using XDCR and eventually indexed.

Key-value data access

While Couchbase is document database, at its heart is a distributed key-value (KV) store. A KV store is an extremely simple, schema-less approach to data management that, as the name implies, stores a unique ID (key) together with a piece of arbitrary information (value); it may be thought of as a hash map or dictionary. The KV store itself can accept any data, whether it be a binary blob or a JSON document, and Couchbase features such as N1QL make use of the KV store's ability to process JSON documents.

Due to their simplicity, KV operations execute with extremely low latency, often sub-millisecond. The KV store is accessed using simple CRUD (Create, Read, Update, Delete) APIs, and provide the simplest interface when accessing documents using their IDs.

The KV store contains the authoritative, most up-to-date state for each item. Query, and other services, provide eventually consistent indexes, but querying the KV store directly will always access the latest version of data. Applications use the KV store when speed, consistency, and simplified access patterns are preferred over flexible query options.

All KV operations are atomic, which means that Read and Update are individual operations. In order to avoid conflicts that might arise with multiple concurrent updates to the same document, applications may make use of Compare-And-Swap (CAS), which is a per-document checksum that Couchbase modifies each time a document is changed.

Query service

The query service is an engine for processing N1QL queries and follows the same scalability paradigm that all the services use which allows, allowing the user to scale query workloads independently of other services as needed.

Non-1st normal form query language (N1QL, pronounced “nickel”) is the first NoSQL query language to leverage the flexibility of JSON with the expressive power of SQL. N1QL is an implementation of the SQL++ standard. N1QL enables clients to access data from Couchbase using SQL-like language constructs, as N1QL’s design was based on SQL. It includes a familiar data definition language (DDL), data manipulation language (DML), and query language statements, but can operate in the face of NoSQL database features such as key-value storage, multi-valued attributes, and nested objects.

N1QL provides a rich set of features that let users retrieve, manipulate, transform, and create JSON document data. Its key features include a powerful SELECT statement that extends the functionality of the SQL SELECT statement to work with JSON documents. Of particular importance are the USE KEYS, NEST, and UNNEST sub-clauses of the FROM clause in N1QL as well as the MISSING boolean option in the WHERE clause.

The following examples illustrate three sample queries – showing common SQL capabilities and the JSON responses.

N1QL supports standard SELECT, FROM, WHERE, GROUP BY clauses as well as JOIN capabilities	`SELECT c.name, o.order_date`
	`FROM customers AS c`
	`LEFT OUTER JOIN orders AS o`
	`  ON c.custid = o.custid`
	`WHERE c.custid = "C41";`
		`{`
		` "results": [`
		`  {`
		`  "name": "R. Duvall",`
		`  "order_date": "2017-04-29"`
		`  },`
		`  {`
		`  "name": "R. Duvall",`
		`  "order_date": "2017-09-02"`
		`  }`
		`  ]`
		`}`

Use UNNEST to extract individual items from a nested JSON array	`SELECT o.orderno,`
	`  i.itemno AS item_number,`
	`  i.qty AS quantity`
	`FROM orders AS o`
	`UNNEST o.items AS i`
	`WHERE i.qty > 100`
		`{`
		`  "results": [`
		`  {`
		`  "orderno": 1002,`
		`  "item_number": 680,`
		`  "quantity": 150`
		`  },`
		`  {`
		`  "orderno": 1005,`
		`  "item_number": 347,`
		`  "quantity": 120`
		`  },`
		`  {`
		`  "orderno": 1006,`
		`  "item_number": 460,`
		`  "quantity": 120`
		`  }`
		`  ]`
		`}"`

Use MISSING boolean keyword in WHERE clause to adapt queries when a schema has changed or lacks specific keys	`SELECT o.orderno, SUM(o.cost) AS cost FROM orders AS o WHERE o.cost IS NOT MISSING GROUP BY o.orderno;`	`{ "results": [ { "orderno": 1002, "cost": 220, }, { "orderno": 1005, "cost": 623, } ] }`

Index service for N1QL

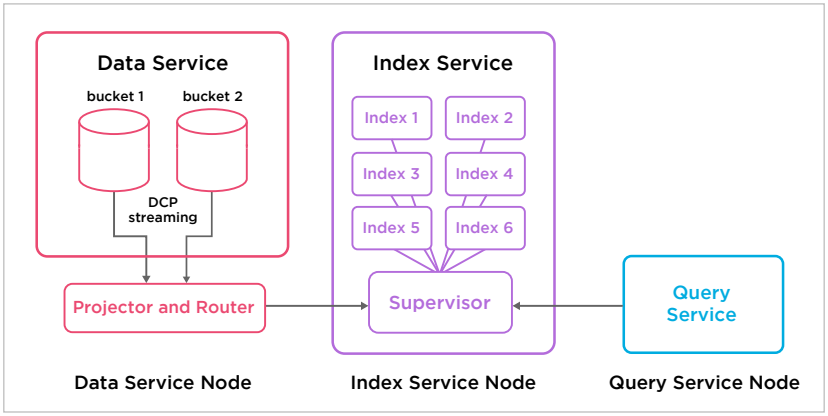
Secondary indexing is an important part of making queries run efficiently and Couchbase provides a robust set of index types and management options. The index service is responsible for all of the maintenance and management tasks of indexes, known as Global Secondary Indexes (GSI). The index service monitors document mutations to keep indexes up to date, using the database change protocol stream (DCP) from the data service. It is distinct from the query service, allowing their workloads to be isolated from one another where needed.

The following is a sample of some of the types of indexes supported by the index service:

- **Primary** – indexes whole bucket on document key
- **Secondary** – indexes a scalar, object, or array using a key-value
- **Composite/Covered** – multiple fields stored in an index
- **Functional** – secondary index that allows functional expressions instead of a simple key-value
- **Array** – an index of array elements ranging from plain scalar values to complex arrays or JSON objects nested deeper in the array
- **Adaptive** – secondary array index for all or some fields of a document without having to define them ahead of time

Query optimization

The query service uses a query optimizer to take advantage of indexes that are available. Index nodes can handle much of the data aggregation pipeline as well, so that less data is sent back to the query node for processing.



Query consistency

Under the hood, Couchbase indexes are updated asynchronously after the data has been changed by the application. In comparison to other database technologies, this allows for much higher write throughput but introduces the possibility of inconsistency between the data and its indexes. Couchbase therefore provides several levels of control over query consistency, allowing the application to choosing between faster queries (ignoring pending mutations) and stronger consistency.

The following consistency levels are specified per-query, allowing for even finer grained and dynamic control of these tradeoffs:

- **not_bounded** (default) – Return the query response immediately, knowing that some data may still be flowing through the system. This consistency level is useful for queries that favor low latency over consistency.
- **at_plus** – Block the query until its indexes have been updated to the timestamp of the last update, knowing that some updates may have happened since but don't need to be consistent. This is for "read-your-own-write" semantics by a single application process/thread.
- **request_plus** – Block the query until its indexes are updated to the timestamp of the current query request. This is a strongly consistent query and is used to maintain consistency across applications/processes.

Indexes are updated as fast as possible, regardless of query consistency requirements. Even a query requesting strong consistency may return extremely quickly if its indexes are not processing large volumes of updates.

Memory-optimized Indexes (MOI)

Memory-optimized indexes use a skiplist structure as opposed to B-tree indexes, optimizing memory consumption and concurrent processing of index updates and scans. MOI provide the most optimized index for processing high-velocity mutations and high-frequency scans. MOI is essentially "in-memory" and therefore requires enough RAM to store all indexes.

Search service

The Search service is an engine for performing Full-Text Searches (FTS) on the JSON data stored within a bucket. FTS lets you create, manage, and query inverted indexes for searching of free-form text within a document. The service provides analyzers that perform several types of operations including multi-language tokenization, stemming, and relevance scoring.

Search nodes incorporate both an indexer and query processor, much like the query and index services, except these don't run on separate nodes – both workloads run on each search node.

As with the other services, data nodes use the DCP stream to send mutations to the FTS indexer process for index updating whenever data changes. Index creation is highly configurable through a JSON index definition file, Couchbase SDK, or through a graphical web interface as part of the administration console.

Documents can be indexed differently depending on a document type attribute, a document ID, or the value of a designated attribute. Each index definition can be assigned its own set of analyzers and specific analyzers can be applied to indexes for a subset of fields.

Indexes are tied to a specific bucket, but it is possible to create virtual index aliases that combine indexes from multiple buckets into a single seamless index. These aliases also allow application developers to build new indexes and quickly change over to new ones without having to take an index offline.

Searching and indexing use the same set of analyzers for finding matching data. All data, when indexed, flows through the analyzer steps as defined by the index. Then search requests are received and passed through the same steps – for example, tokenization, removing stop words, and stemming terms. These analyzed search requests are then looked up by the indexer in the index and matches are returned. The results include the source request, list of document IDs, and relevance scoring.

Other indexing and search-time options provide fine-grained control over indexing more or less information depending on the use case. For example, text snippets may also be stored in the index and included in the search response so that retrieving full documents from the data service is not required.

Couchbase developed Bleve, the open source Go-based search project, for the FTS capabilities, including language support, scoring, etc.

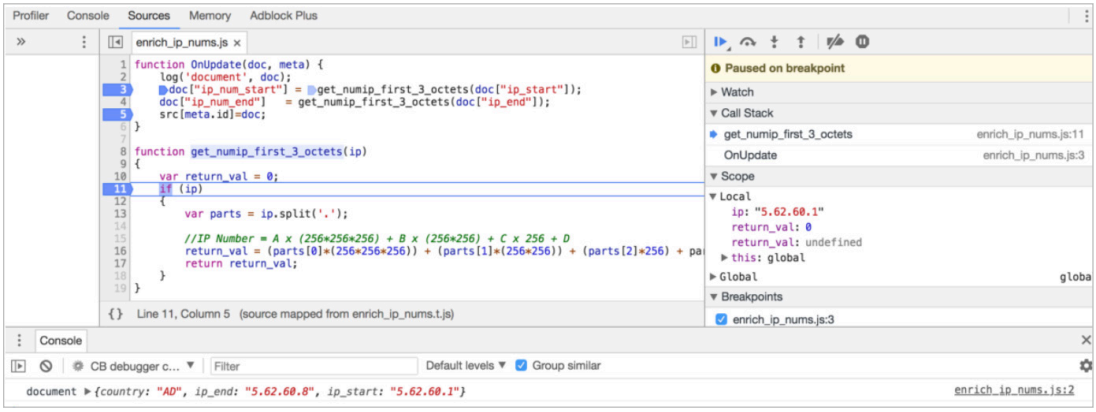
Eventing service

The eventing service supports custom server-side functions (written in JavaScript) that are automatically triggered using an Event-Condition-Action model. These functions receive data from the DCP stream for a particular bucket and execute code when triggered by data mutations. Similar to other services, the eventing service scales linearly and independently.

Code processes the source data and commits it as a new or updated document in another bucket.

The core of eventing functions is a Google V8 execution container. Functions inherit support for most of the standard ECMAScript constructs that are available through V8. Some capabilities have been removed to support the ability to shard and scale execution automatically. Additionally, to optimize language utilization of the server environment, some new constructs have been added.

Code for functions is written in a web-based JavaScript code editor and features an extensive in-browser debugging environment.

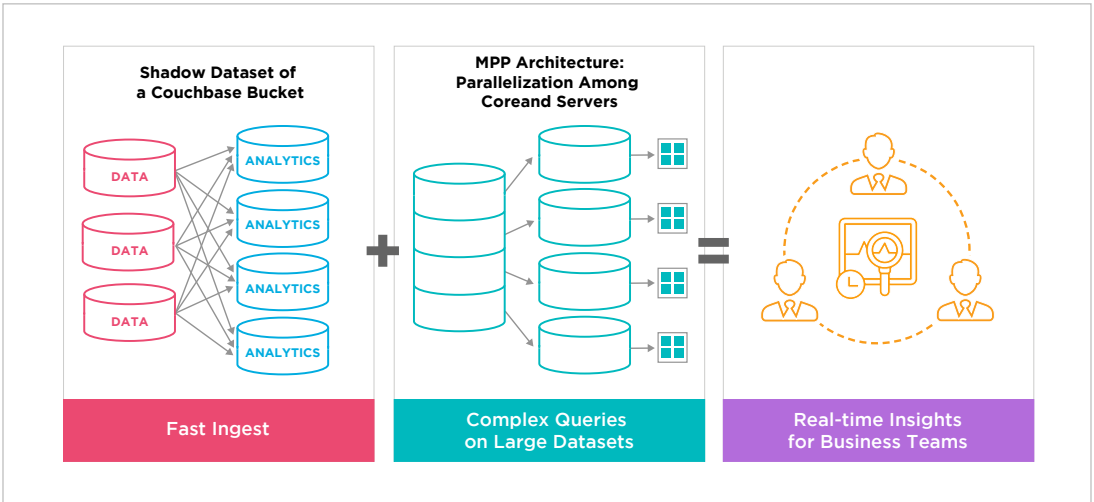


Analytics

The analytics service provides an ad hoc querying capability without the need for indexes, bringing a hybrid operational and analytical processing (HOAP) model for real-time and operational analytics on the JSON data within Couchbase. It uses the same N1QL language as the query service.

It is designed to efficiently run complex queries over a large number of documents, including query features such as ad hoc join, set, aggregation, and grouping operations. In a typical operational or analytical database, any of these kinds of queries may result in inefficiencies: long running queries, I/O constraints, high memory consumption, and/or excessive network latency due to data fetching and cross-node coordination.

Because the service supports efficient parallel query processing and bulk data handling, and runs on separate nodes, it is often preferable for expensive queries, even if the queries are predetermined and could be supported by an operational index. With traditional database technology it is important to segregate operational and analytic workloads. This is usually done by batch exporting of data from operational databases into an analytic database that does further processing. Couchbase provides both the operational database as well as a scalable analytics database – all in one NoSQL platform.



Data is pushed from the DCP stream into what are known as shadow buckets – copies of data that are processed and immediately ready for analysis by a dedicated massively parallel processing (MPP) analytics engine. As shadowed data is linked directly to the operational data in real time, queries do not affect performance of that operational data. You can add more analytics nodes to reduce analytics query time.

The Couchbase Analytics approach has significant advantages compared to the commonly employed alternatives:

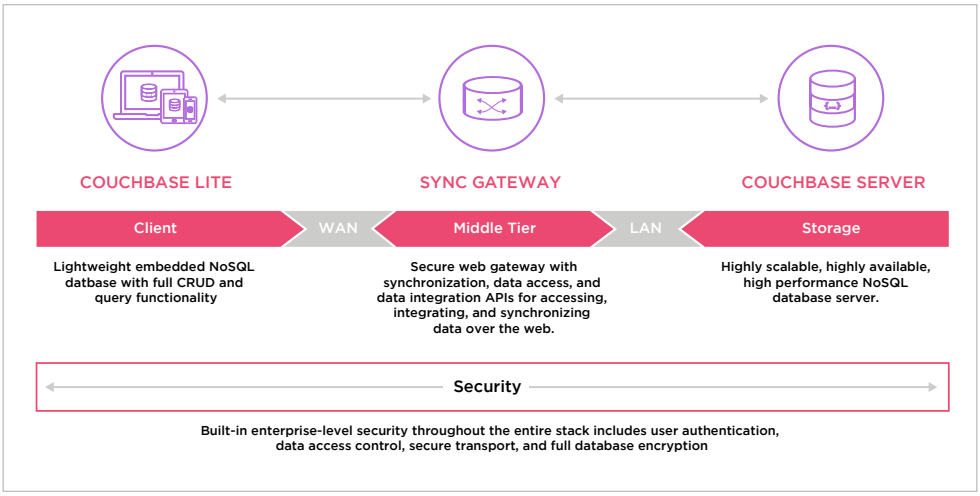
Common data model – supports the same flexible document data model and schema used for operational data with no transformation required for analysis.

Workload isolation – operational query latency and throughput are protected from slowdowns due to your analytical query workload – but without the complexity of operating a separate analytical database.

High data freshness – the DCP stream provides a fast memory-to-memory protocol that nodes use to synchronize data among themselves, allowing analytics to run on data that's extremely current, without extract/load or other hassles and delays.

Mobile

Couchbase Mobile provides the ability to have Couchbase data brought to, and communicate with, edge devices in an efficient and secure manner. It consists of two primary components: Couchbase Lite and Sync Gateway. They work together to provide a robust, always-on, mobile database solution that works on top of Couchbase Server.



Couchbase Lite is the embedded database that handles the query and data management functionality you would expect from a database, but on mobile devices. The client platform closely integrates with the Sync Gateway API for communications, which takes care of the security and synchronization of the data over the network to the server. As workloads and the number of edge devices increase, more Sync Gateway nodes can be deployed to help maintain high performance.

Couchbase Lite has several important features that help to build a robust and powerful application on the edge, including:

Offline use – allowing applications to be always-on, able to run and store data **on device** until the network becomes available again.

Peer-to-peer replication – providing synchronization capability **between devices** using Bluetooth and other network protocols even when offline from the primary database.

On-device data encryption – keeping data secure, especially when devices are lost or stolen.

DISTRIBUTED FOUNDATION

The foundation of Couchbase is a clustering approach that provides flexible options for scaling while maintaining performance and availability. This scaling approach is applied across all levels of the cluster, providing high-performance flexibility for nodes, services, buckets, and vBuckets.

Node-level architecture

A Couchbase cluster consists of a group of interchangeable, largely self-sufficient nodes that operate in a peer-to-peer topology. There is just one Couchbase node type, though the services running on that node can be managed as required (see Multi-Dimensional Scaling).

Having a single-node type greatly simplifies the installation, configuration, management, and troubleshooting of a Couchbase cluster, both in terms of what you must do as a human operator and what the automatic management needs to do. There is no concept of master nodes, slave nodes, config nodes, name nodes, or head nodes.

Components of a Couchbase node include the cluster manager and, optionally, the data, query, index, analytics, search, and eventing services. There is also the underlying managed cache and storage components. By dividing up potentially conflicting workloads in this way, a Couchbase node can achieve maximum throughput and resource utilization and minimum latency.

Nodes can be added or removed easily through a rebalance process, which redistributes the data evenly across all nodes. The rebalance process is done online and requires no application downtime, and can be initiated at the click of a button or one command on the command line.

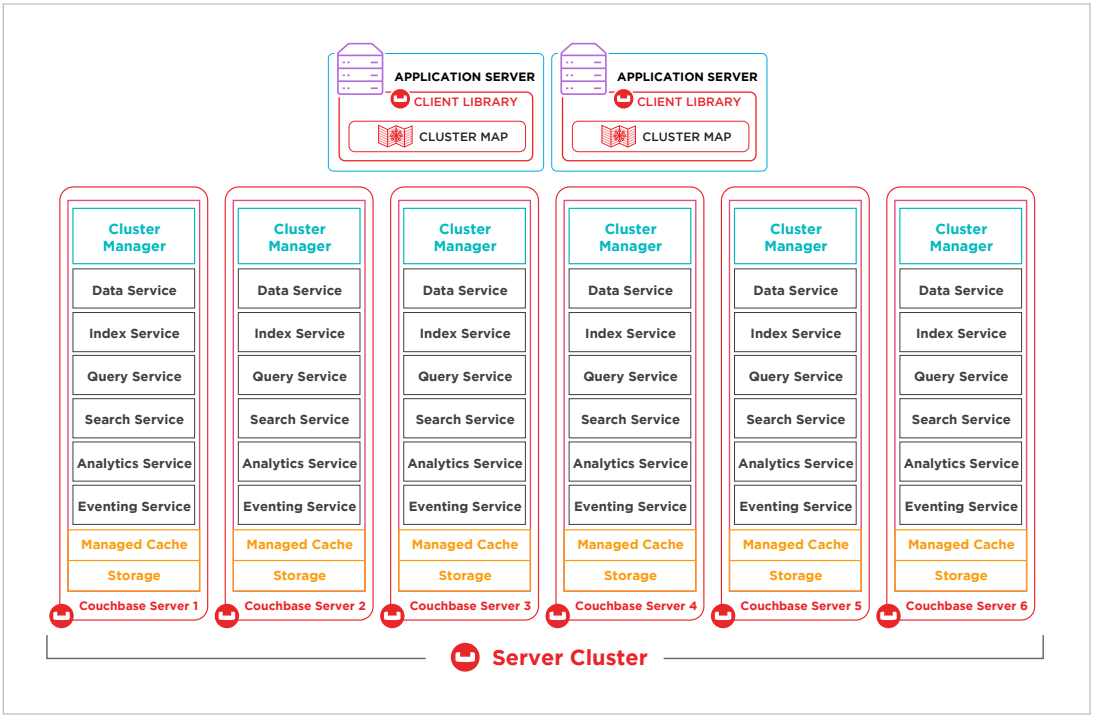
Couchbase Server is typically deployed on a cluster of commodity servers, although for development purposes all functionality can be run on a single node. An application can be developed on a small scale, even on a laptop, and then deployed to a distributed cluster without any architecture or behavioral changes to that application.

Capacity can be increased or decreased simply by adding or removing nodes. In this way a cluster can grow CPU, RAM, disk, and network capacity by adding physical servers or virtual machines that have the exact same software installed.

Cluster architecture

A cluster consists of one or more instances of Couchbase Server, each running on an independent node. Data and services are shared across the cluster.

The following figure shows application servers communicating with the cluster overall, but because they are also aware of the individual node topology, they can adapt as needed. For example, if a replica needs to be read, the application server can access it directly because it knows about the overall cluster configuration.



The various services that Couchbase provides are also fed data through the DCP stream, which is internally used for sending new/changed data to these services as well as providing the basis for keeping data in sync between nodes in the cluster.

Cluster/node configuration

When Couchbase is being configured on a node, it can be specified either as its own, new cluster, or as a participant in an existing cluster. Thus, once a cluster exists, successive nodes can be added to it. When a cluster has multiple nodes, the Couchbase cluster manager runs on each node: this manages communication between nodes, and ensures that all nodes are healthy.

Services can be configured to run on all or some nodes in a cluster, and can be added/removed as warranted by established performance needs. For example, given a cluster of five nodes, a small dataset might require the data service on only one of the nodes; a large dataset might require four or five. Alternatively, a heavy query workload might require the query service to run on multiple nodes, rather than just one. This ability to scale services individually promotes optimal hardware resource utilization.

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 cluster-wide 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 failover through external monitoring scripts 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.

In addition to the cluster orchestrator, there are three primary cluster manager components on each Couchbase node:

The heartbeat watchdog – periodically communicates with the cluster orchestrator using the heartbeat protocol, providing regular health updates for the server. If the orchestrator crashes, existing cluster server nodes will detect the failed orchestrator and elect a new orchestrator.

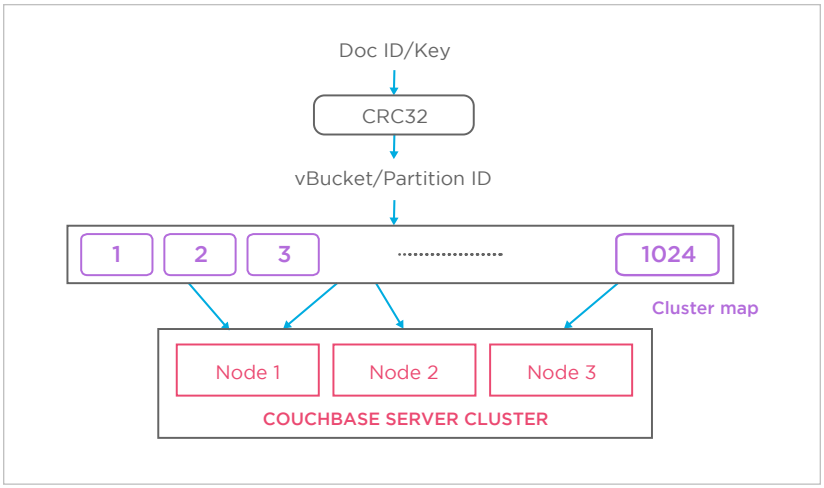
The process monitor – monitors local data manager activities, restarts failed processes as required, and contributes status information to the heartbeat process.

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.

Client connectivity

To talk to all the services of a cluster, applications use the Couchbase SDK. Support is available for a variety of languages including Java, .NET, PHP, Python, Go, Node.js, and C/C++. These clients are continually aware of the cluster topology through cluster map updates from the cluster manager. They automatically send requests from applications to the appropriate nodes for KV access, query, etc.

When creating documents, clients apply 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. Because a common hash function is used, it is always possible for a client to determine on which node the source document can be found.



Topology-aware client

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. 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 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.
4. When a document arrives in a cluster, Couchbase Server replicates the document, caches it in memory and asynchronously stores it on disk.

Data transport via Database Change Protocol (DCP)

DCP is the protocol used to stream bucket-level mutations. It is used for high-speed replication of data as it mutates – to maintain replica vBuckets, global secondary indexes, full-text search, analytics, eventing, XDCR, and backups. Connectors to external services, such as Elasticsearch, Spark, or Kafka are also fed from the DCP stream.

DCP is a memory-based replication protocol that is *ordering*, *resumable*, and *consistent*. DCP stream changes are made in memory to items by means of a *replication queue*.

An external application client sends the operation requests (read, write, update, delete, query) to access or update data on the cluster. These clients can then receive or send data to DCP processes running on the cluster. External data connectors, for example, often sit and wait for DCP to start sending the stream of their data when mutations start to occur.

Whereas an internal DCP client, used by the cluster itself, streams data between nodes to support replication, indexing, cross datacenter replication, incremental backup, and mobile synchronization. Sequence numbers are used to track each mutation in a given vBucket, providing a means to access data in an ordered manner or to resume from a given point in time.

Multi-Dimensional Scaling (MDS)

Couchbase MDS features improve performance and throughput for mission-critical systems by enabling independent scaling of data, query, and indexing workloads. Scale-out and scale-up are the two scalability models typical for databases – Couchbase takes advantage of both. There are unique ways to combine and mix these models in a single cluster to maximize throughput and latencies. With MDS, admins can achieve both the existing homogeneous scalability model and the newer independent scalability model.

Homogeneous scaling model

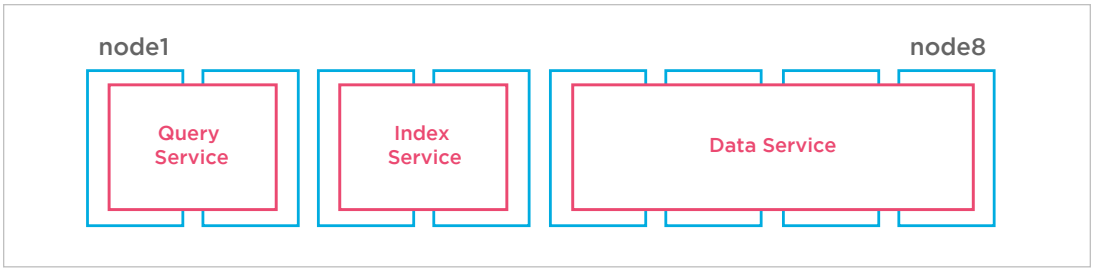
To better understand the multi-dimensional scaling model, it is beneficial to take a look at a typical homogeneous scaling model.

In this model, application workloads are distributed equally across a cluster made up of the homogeneous set of nodes. Each node that does the core processing takes a similar slice of the work and has the same hardware resources.

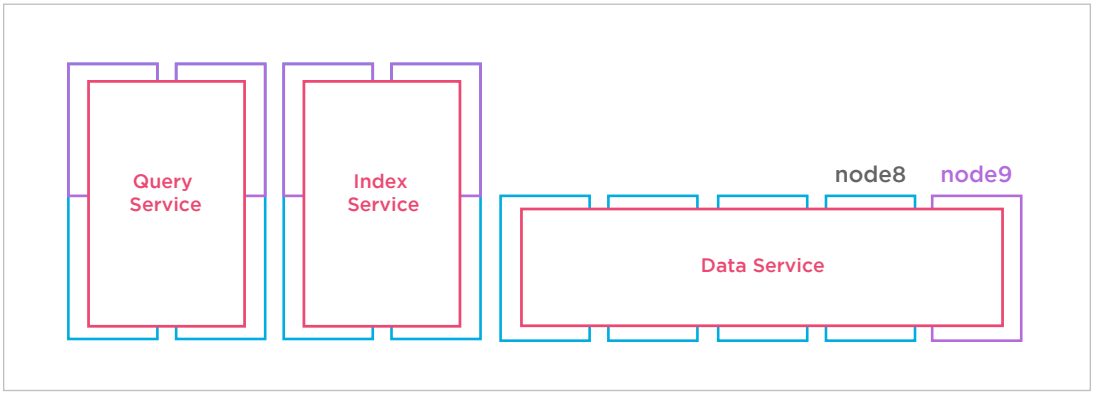
This model is available through MDS and is simple to implement but has a couple drawbacks. Components processing core data operations, index maintenance, or executing queries all compete with each other for resources. It is impossible to fine-tune each component because each of them has different demands on hardware resources. This is a common problem with other NoSQL databases. While the core data operations can benefit greatly from scale-out with smaller commodity nodes, many low latency queries do not always benefit from wider fan-out.

Independent scaling model

MDS is designed to minimize interference between services. When you separate the competing workloads into independent services and isolate them from each other, interference among them is minimized. The figure below demonstrates a deployment topology that can be achieved with MDS. In this topology, each service is deployed to an independent zone within the cluster.



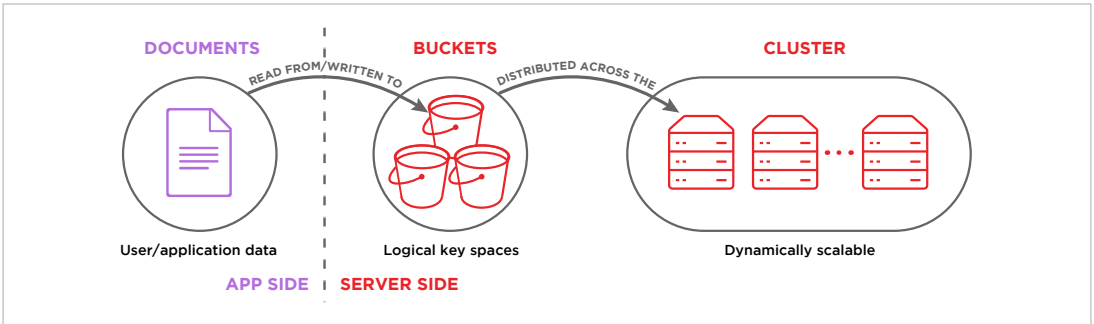
Each service zone within a cluster (data, query, and index services) can now scale independently so that the best computational capacity is provided for each of them.



In the figure above, the green additions signify the direction of scaling for each service. In this case, query and index services scale-up over the fewer sets of powerful nodes and data service scales out with an additional node.

Data distribution

Couchbase partitions data into vBuckets (synonymous to shards or partitions) to automatically distribute data across nodes, a process sometimes known as auto-sharding.



vBuckets help enable data replication, failover, and dynamic cluster reconfiguration. Unlike data buckets, users and applications do not manipulate vBuckets directly. Couchbase automatically divides each bucket into 1024 active vBuckets and 1024 replica vBuckets per replica, and then distributes them evenly across the nodes running the data service within a cluster. vBuckets do not have a fixed physical location on nodes; therefore, there is a mapping of vBuckets to nodes known as the cluster map. Through the Couchbase SDK, the application automatically and transparently distributes the data and workload across these vBuckets.

Index partitions and replicas

Global secondary indexes (GSI) can also be partitioned using several approaches depending on the pattern of queries that need to be supported. Commonly, GSI are stored identically on each index node so that any node can help with queries. But in some cases it is better for performance or storage management to spread the indexes across several nodes. There are many options for creating these partitions – different attributes in a document can be used as a hash key (one or more), partitions can also be assigned to specific nodes, and more.

As documents are added and updated, those mutations are streamed from memory to a local projector process which then forwards them to the relevant index service node(s). Every index node has a supervisor process running the index service. This process listens to changes from the projector processes on the data nodes. It evaluates the incoming stream of changes for the specific indexes created on the node running the index service. Each index is then updated independently with that data.

Individual indexes can be automatically replicated to other nodes in the cluster to achieve high availability, ensuring that an index continues to function even if a node hosting the index is unavailable. Queries will load balance across the indexes and if one of the indexes become unavailable, all requests are automatically rerouted to the available remaining index without application or admin intervention.

Similarly, Couchbase supports independent partitioning of indexes to distribute the data volumes and load of a single index across multiple processes and/or nodes.

Partitioning other services

All Couchbase services partition data and workloads across available nodes. Full-text search, eventing, and analytics services partition and replicate data and processes. They also all redistribute these across new cluster topologies when rebalancing occurs.

- **Search service** – automatically partitions its indexes across all search nodes in the cluster, ensuring that during rebalance, the distribution across all nodes is balanced.
- **Eventing service** – vBucket processing ownership is distributed across eventing nodes.
- **Analytics service** – a single copy of all analytics data is partitioned across all cluster nodes that run the service.

Rebalancing the cluster

When the number of servers in the cluster changes due to scaling out or node failures, data partitions must be redistributed. This ensures that data is evenly distributed across the cluster, and that application access to the data is load balanced evenly across all the servers. This process is called rebalancing. All Couchbase services are rebalance-aware and follow their own set of internal processes for rebalance as needed.

Rebalancing is triggered using an explicit action from the admin web UI or through a REST call. When initiated, 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 three 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 can 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.

High availability

To meet the demands of high availability requirements, all Couchbase operations can be done while the system remains online, without requiring modifications or interrupting running applications. The system never needs to be taken offline for routine maintenance such as software upgrades, index building, compaction, hardware refreshes, or any other operation. Even provisioning or removing nodes can be done entirely online without any interruption to running applications, and without requiring developers to modify their applications.

Built-in fault tolerance mechanisms protect against downtime caused by arbitrary unplanned incidents, including server failures. Replication and failover are important mechanisms that increase system availability. Couchbase replicates data across multiple

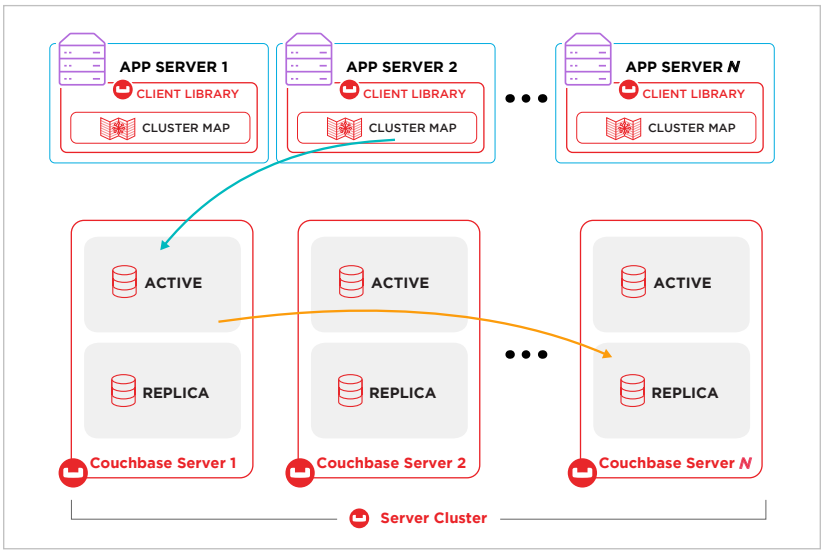
nodes to support failover. Ensuring that additional copies of the data are available is automated to deal with the inevitable failures that large distributed systems are designed to recover from. All of this is done automatically without need for manual intervention or downtime.

Intra-cluster replication

Up to three replica buckets can be defined for every bucket. Each replica itself is also implemented as 1024 vBuckets. A vBucket that is part of the original implementation of the defined bucket is referred to as an active vBucket. Therefore, a bucket defined with two replicas has 1024 active vBuckets and replica vBuckets. Typically, only active vBuckets are accessed for read and write operations: although vBuckets are able to support read requests. Nevertheless, vBuckets receive a continuous stream of mutations from the active vBucket by means of DCP, and are thereby kept constantly up to date.

To ensure maximum availability of data in case of node failures, the master services for the cluster calculate and implement the optimal vBucket distribution across available nodes: consequently, the chance of data loss through the failure of an individual node is minimized, since replicas are available on the nodes that remain.

Active and replica vBuckets correspond to a single, user-defined bucket, for which a single replication instance has been specified. No replica resides on the same node as its active equivalent.



When a node becomes unavailable, failover can be performed and the cluster manager is instructed to read and write data only on available nodes. Failover can be performed by manual intervention, or automatically, promoting replica vBuckets to active status when needed. Automatic failover can be performed only for one node at a time, and only up to a configurable number of times, the maximum being three.

The cluster manager never performs automatic failover where data loss might result. The number of times failover can be safely performed depends on how many nodes and replicas exist. For example, in a five-node cluster with one replica, a single node can be failed over without danger, but if a second node fails, failover might result in data loss, due to required replicas no longer being available. Similarly, in a five-node cluster with two replicas, two nodes can be failed over without danger, a third cannot. In such cases, a slower, manual recovery process is required.

Node failover

Failover is the process in which a node of a Couchbase cluster is removed quickly as opposed to intentional removal and rebalancing.

Auto-failover allows unresponsive servers to be failed over automatically by the cluster manager. Data partitions in Couchbase are always served from a single master node. As a result, if that node is down, the data will not be available until restored. The server will either need to be manually or automatically failed over in order to promote replica data partitions on replica servers to active data partitions so that they can be accessed by the application.

The administrator will not always be able to manually fail servers over quickly enough to avoid application downtime, so Couchbase provides an auto-failover feature. This feature allows the cluster manager to automatically fail over nodes that are down and bring the cluster back to a healthy state as quickly as possible.

In Couchbase Server Enterprise Edition nodes can also be automatically failed over when the data service reports sustained disk I/O failures.

Failover choices

As a node failover has the potential to reduce the performance of your cluster, you should consider how best to handle a failed node situation and also size your cluster to plan for failover.

Manual or monitored failover
Manual failover is performed by either human monitoring or by using a system external to the cluster. An external monitoring system can monitor both the cluster and the node environment so that you can make a more data-driven decision.

Human intervention
Humans are uniquely capable of considering a wide range of data, observations, and experiences to resolve a situation in the best possible way. Many organizations disallow automated failover because they want a human to consider the implications. Human intervention tends to be slower than using a computer-based monitoring system.

External monitoring
Another option is to have a system monitoring the cluster via the Couchbase REST API. Such an external system can failover nodes successfully because it can take into account system components that are outside the scope of Couchbase Server.

For example, monitoring software can observe that a network switch is failing and that there is a dependency on that switch by the Couchbase cluster. The system can determine that failing nodes will not help the situation and will, therefore, not failover the node. The monitoring system can also determine if the components around Couchbase Server are functioning and if the various nodes in the cluster are healthy.

If the monitoring system determines the problem is only with a single node and remaining nodes in the cluster can support aggregate traffic, then the system may safely failover the node using the REST API or command-line tools.

Automatic failover
The cluster manager handles the detection, determination, and initiation of the processes to failover a node without user intervention. Once the problem has been identified and fixed, it still requires you to initiate a rebalance to return the cluster to a healthy state.

Server group awareness

Server group awareness provides enhanced availability. Specifically, it protects a cluster from large-scale infrastructure failure through the definition of *groups*. Each group is created by an appropriately authorized administrator, and specified to contain a subset of the nodes within a Couchbase cluster. Following group definition and rebalance, the active vBuckets for any defined bucket are located on one group, while the corresponding replicas are located on another group. This allows group failover to be enabled, so that if an entire group goes offline, its replica vBuckets, which remain available on another group, can be automatically promoted to active status.

Groups should be defined in accordance with the physical distribution of nodes. For example, a group should only include the nodes that are in a single server rack, or in the case of cloud deployments, a single availability zone. Thus, if the server rack or availability zone becomes unavailable due to a power or network failure, group failover can allow continued access to the affected data.

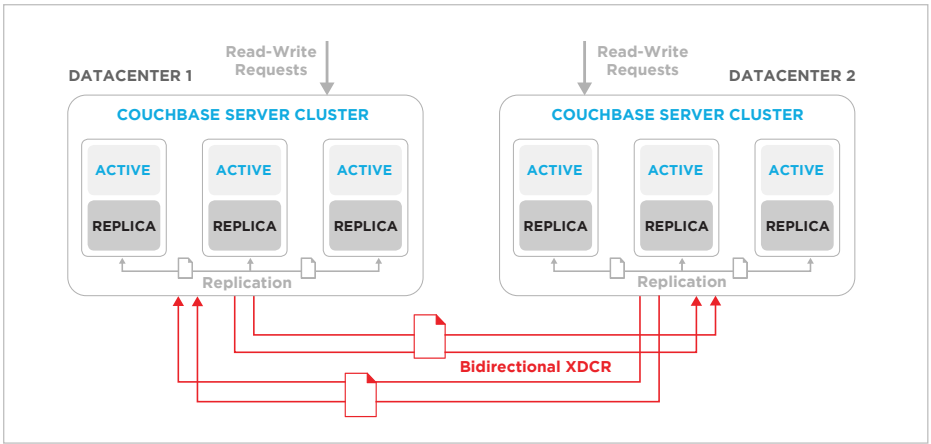
Data protection is optimal when groups are assigned equal numbers of nodes, and vBuckets are therefore distributed such that none ever occupies the same group as its associated active vBucket. By contrast, when groups are not assigned equal numbers of nodes, rebalance can only produce a best effort redistribution of replica vBuckets. This may result in replica vBuckets occupying the same group as their associated active vBuckets; meaning that data may be lost if such a group becomes unavailable.

Server group awareness considerations:

- Server group awareness only applies to the data service.
- Failover should be enabled for server groups only if three or more server groups have been established, and sufficient capacity exists to absorb the load of any failed-over group.
- The first node, and all subsequent nodes, are automatically placed in a server group named Group 1. Once you create additional server groups, you are required to specify a server group when adding additional cluster nodes.

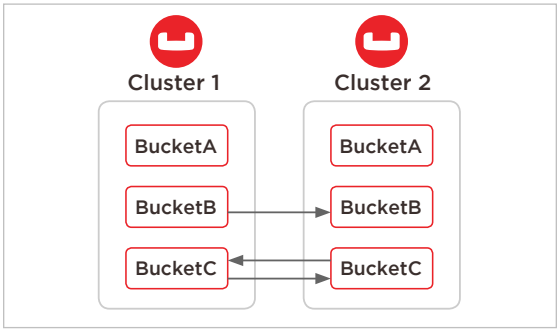
Cross Datacenter Replication (XDCR)

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.



XDCR and intra-cluster replication occurs simultaneously. For example, 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 managed object cache so that replica data on the destination cluster can undergo low latency read/write operations.

XDCR can be set up 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. Bidirectional XDCR can be set up between Bucket C on both Cluster 1 and 2. There is unidirectional XDCR between Bucket B on both clusters. Bucket A is not replicated.



XDCR provides only a single basic mechanism from which replications are built: this is the *unidirectional* replication. A *bidirectional* topology is created by implementing two *unidirectional* replications, in opposite directions, between two clusters; such that a bucket on each cluster functions as both source and target.

Used in different combinations, unidirectional and bidirectional replication can support complex topologies; an example being the *ring* topology, where multiple clusters each connect to exactly two peers, so that a complete ring of connections is formed.

When a bucket is specified as the source for an XDCR replication, all data in the bucket is replicated. Thus, if replication is started between source and target buckets that initially contain different datasets, the replication process eventually establishes a complete superset of data within each bucket.

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.

The source and destination clusters can have a different number of servers in varying topologies. 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.

Conflict resolution

Within a cluster, Couchbase provides strong consistency at the document level. Likewise, XDCR also provides eventual consistency across clusters. Built-in conflict resolution will pick the same "winner" on both clusters if the same document was mutated on both 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.

Security

Couchbase Server can be rendered highly secure, so as to preserve the privacy and integrity of data, and account for access attempts. Couchbase provides the following security facilities:

- **Authentication** – All administrators, users, and applications (all formally considered users) must authenticate in order to gain server access. Users can be authenticated by means of either the local or an external password registry. Authentication can be achieved by either passing credentials directly to the server, or by using a client certificate, in which the credentials are embedded. Connections can be secured by means of SCRAM and TLS.
- **Authorization** – Couchbase Server uses Role-Based Access Control (RBAC) to associate users with specifically assigned roles, each role corresponds to system-defined privileges, which allow degrees of access to specific system resources. During authentication, user roles are determined; therefore, authorization is granted if the role has approved system access, otherwise it is denied.
- **Auditing** – Actions performed on Couchbase Server can be audited. This allows administrators to ensure that system management tasks are being appropriately performed.

Couchbase Server supports the encryption of data that is at rest on disk, on the wire, and held by applications. Additionally, it provides a system of secret management, which allows essential information to the security and maintenance of Couchbase Server to be stored in encrypted form; and then decrypted and appropriately used at cluster startup.

Encryption at rest

Couchbase can run on top of commonly used third-party encryption tools, such as Linux Unified Key Setup (LUKS), Vormetric, Gemalto, and Protegrity, to provide a complete solution for data encryption at rest while also being accessible to all Couchbase services.

Couchbase SDKs provide methods for encrypting portions of documents for added security at the application layer. Encrypted data sent back to the server is then always stored in that encrypted state, at rest, and is inaccessible without the key used by the original application layer. Couchbase cannot index or query application-encrypted fields.

Encryption over-the-wire

For an application to communicate securely with Couchbase Server, SSL/TLS must be enabled on the client side. When a TLS connection is established between a client application and Couchbase Server (running on port 18091), a handshake occurs, as defined by the TLS Handshake Protocol. As part of this exchange, the client must send to the server a cipher-suite list; which indicates the cipher-suites that the client supports, in order of preference. The server replies with a notification of the cipher-suite it has duly selected from the list. Additionally, symmetric keys to be used by client and server are selected by means of the RSA key-exchange algorithm.

Couchbase SDKs support SSL/TLS encryption and must use the Couchbase network port 11207 for secure communication. Couchbase Server uses ciphers that are accepted by default by OpenSSL. The default behavior employs high-security ciphers, built into OpenSSL, but can be overridden depending on the security level required by the cluster.

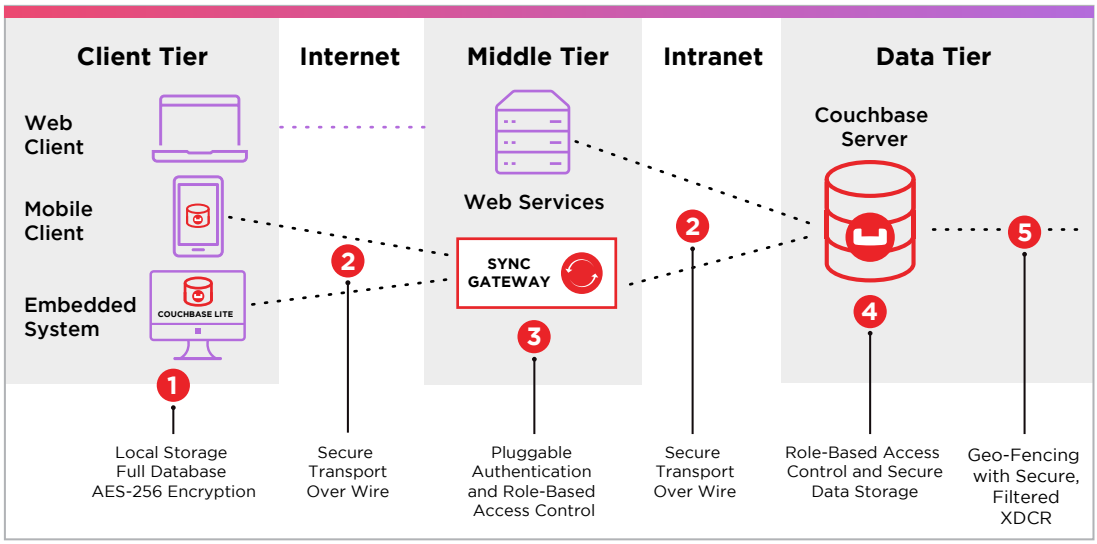
Moving data between nodes

Couchbase Server replicates data across a cluster to ensure high availability of data. When encrypting documents, replica copies are duly transmitted and stored in encrypted form.

For added security, use IPSec on the network that connects the nodes. Note that IPSec has two modes: tunnel and transport. Transport mode is recommended, as it is the easier of the two to set up, and does not require the creation of tunnels between all pairs of Couchbase nodes.

Moving data between datacenters

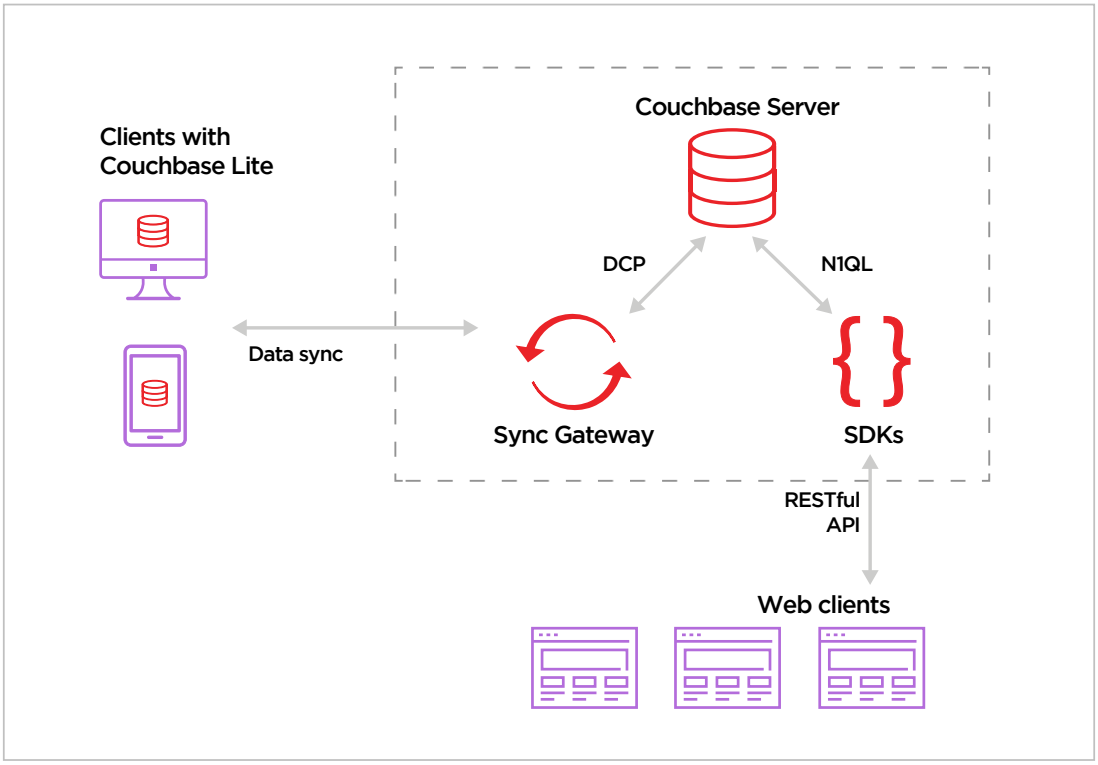
To protect data transmitted between datacenters, TLS is used to encrypt XDCR connections. When TLS in XDCR is enabled, Couchbase uses TLS certificates. TLS versions 1.0 to 1.2 are supported. All traffic between source and destination datacenters is encrypted.



Mobile client synchronization

Couchbase clusters can be extended to mobile applications running on edge devices. This is done by adding Sync Gateway as a middle layer between device applications and Couchbase Server. Couchbase Lite's SDK is then used to develop mobile applications communicating via this synchronization layer.

Using the Sync Gateway service, data can be seamlessly extended to connect with remote edge devices that are occasionally disconnected or connected. Sync Gateway monitors data changes and maintains synchronization between Couchbase Server and mobile applications.



Sync Gateway provides the facility to ensure that all writes happen. Along with security and replication, metadata is managed by Sync Gateway and abstracted from applications reading and writing data directly to Couchbase Server. Sync Gateway uses a feature of Couchbase Server called extended attributes (XATTRs) to store that metadata in an external document fragment. Mobile, web, and desktop applications can therefore write to the same bucket in a Couchbase.

Conflict resolution

As Couchbase Mobile can handle several kinds of scenarios where multiple users may make updates – e.g., offline, peer-to-peer, and live database sync modes – being able to manage mutation conflicts is essential. The system, therefore, keeps track of the changes and prevents new documents from simply overriding others. The developer is given tools to control the save and delete operations so that a last-write-wins or last-write-fails protocol can automatically resolve any conflicts.

RESOURCES

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.

Couchbase has a rich ecosystem of adapters that support other systems:

- SQL integration resources include: CData, Knowi, Talend, and more
- Big data integration includes Spark, Kafka, and Elasticsearch

Several flexible deployment options are available to support different environments:

- Download Couchbase Server for free, visit http://www.couchbase.com/downloads
- Couchbase Autonomous Operator for Kubernetes and OpenShift
- Docker Hub hosts official Couchbase Docker images
- Managed cloud service

Training is available through in-class instructor led and free online courses. See the course catalog at: https://learn.couchbase.com/.

Comprehensive Couchbase documentation is available at: https://docs.couchbase.com/.

About Couchbase

Couchbase's mission is to be the data platform that revolutionizes digital innovation. To make this possible, Couchbase created the world's first Engagement Database. Built on the most powerful NoSQL technology, the Couchbase Data Platform offering includes Couchbase Server and Couchbase Mobile and is open source. The platform provides unmatched agility and manageability – as well as unparalleled performance at any scale – to deliver ever-richer and ever-more-personalized customer experiences.

3250 Olcott Street | Santa Clara, California 95054 | 1-650-417-7500 | www.couchbase.com