Couchbase

# Couchbase Data Platform - Architecture

## Couchbase as Key-Value Store vs. Document Store

Couchbase is capable of storing multiple data types

- Simple data types such as string, number, datetime, and boolean
- Arbitrary binary data
- For most of the simple data types, Couchbase offers a scalable, distributed data store that provides both key-based access as well as minimal operations on the values

Document databases encapsulate stored data into “JSON documents” that they can operate on

- A document is simply an object that contains data in some specific format.
- For example, a JSON document holds data encoded in the JSON format

## The Couchbase Data Model provides a lightweight, flexible schema

# JSON documents

- Data model based on JSON
  - Supports basic and complex data types
  - Rapid serialization and deserialization
  - Extremely convenient for web-application programming
- A document often represents a single instance of an object
  - Equivalent to a row in a relational table
  - Document's attributes equivalent to a column.
  - Couchbase can store JSON documents with varied schemas
- Documents can contain nested structures.
  - Many-to-many relationships without requiring a reference or junction table
  - Naturally expressive of hierarchical data

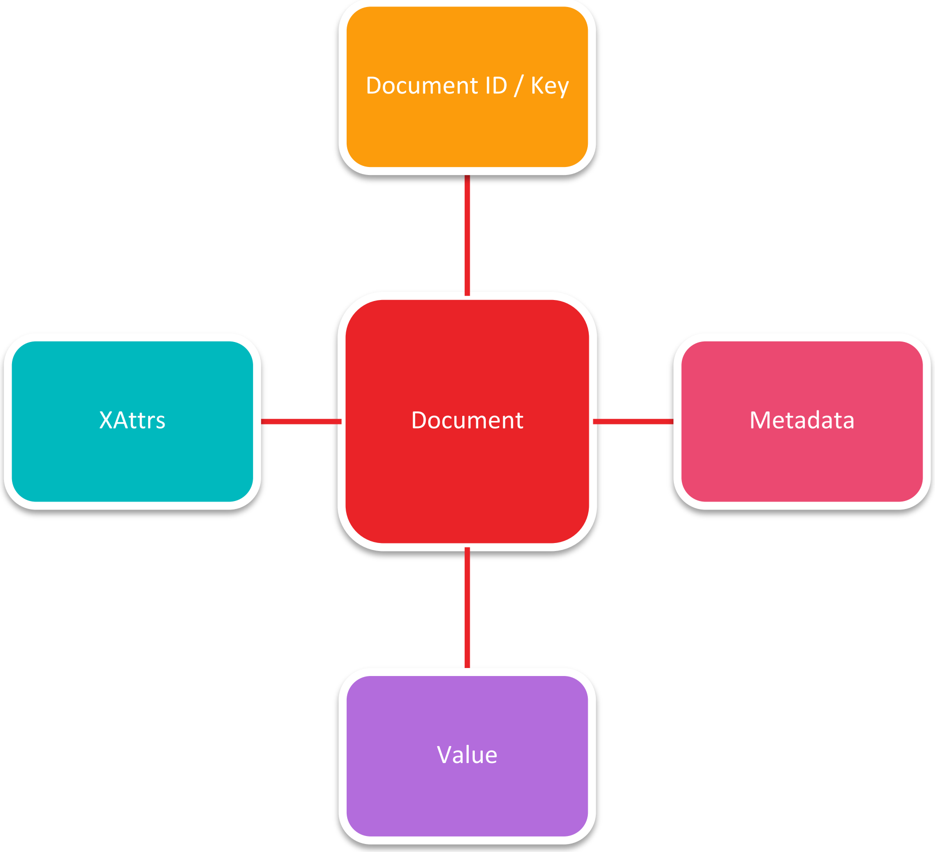
**Simple attributes**

```
{
"firstName":"Javier",
"lastName":"Hernandez",
"skills": [
  "Big Data", "Java", "NoSQL"
],
"experience": [
   {
     "role":"Solution Architect",
     "company":"AppMax"
   },
   {
     "role":"Solution Engineer",
     "company":"Couchbase"
   }
]
}
```

**Complex attributes**

JSON document sample

# Document Structure



**Document ID / Key (Max 250 bytes):**
- Must be unique / Lookup is extremely fast
- Similar to primary keys in relational databases
- Documents are partitioned based on the document ID

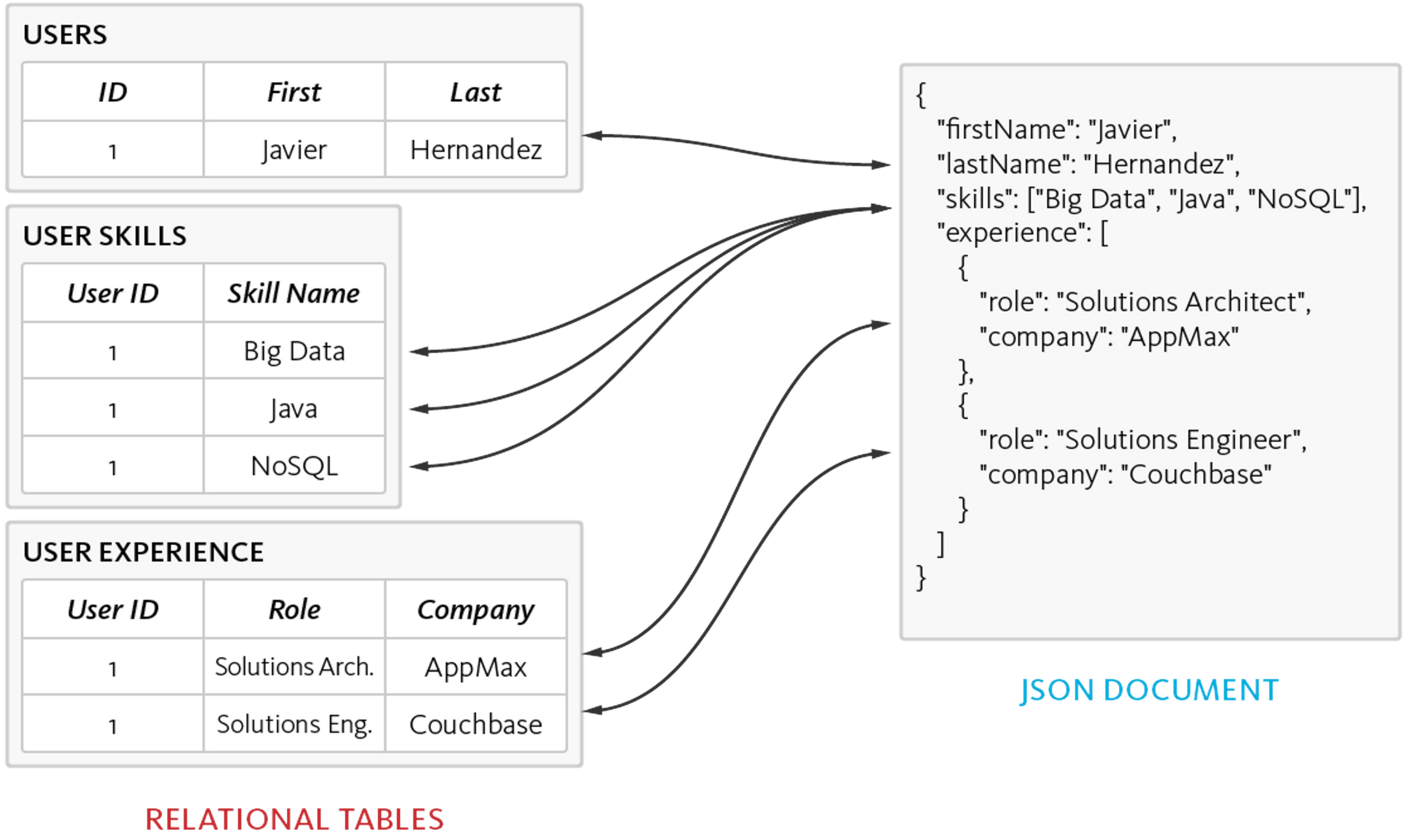
**Metadata (Fixed 56 bytes)**
- CAS Value (unique identifier for concurrency)
- TTL
- Flags (optional client library metadata)
- Revision ID #

**Value (Max 20 MB)**
- JSON
- Binary - integers, strings, booleans
- Common binary values include serialized objects, compressed XML, compressed text, encrypted values

**XAttr (Max 20 MB)**
- Non-enumerable e**X**tended **Attr**ibutes

# Denormalization

Each document can be self-contained for better scalability and latency



USERS

| ID | First | Last |
|---|---|---|
| 1 | Javier | Hernandez |

USER SKILLS

| User ID | Skill Name |
|---|---|
| 1 | Big Data |
| 1 | Java |
| 1 | NoSQL |

USER EXPERIENCE

| User ID | Role | Company |
|---|---|---|
| 1 | Solutions Arch. | AppMax |
| 1 | Solutions Eng. | Couchbase |

RELATIONAL TABLES

```
{
  "firstName": "Javier",
  "lastName": "Hernandez",
  "skills": ["Big Data", "Java", "NoSQL"],
  "experience": [
    {
      "role": "Solutions Architect",
      "company": "AppMax"
    },
    {
      "role": "Solutions Engineer",
      "company": "Couchbase"
    }
  ]
}
```

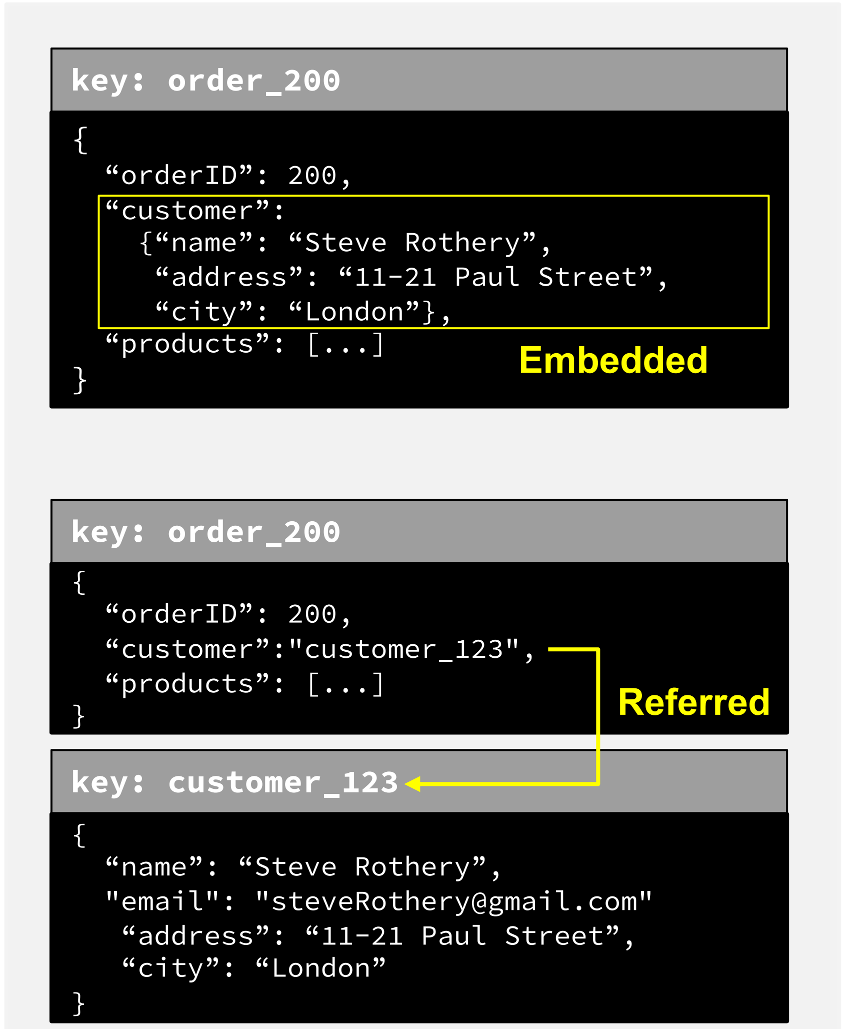
JSON DOCUMENT

## Document Design considerations

### Embedded or Referenced

**Performance and scalability should be considered since design phase**

- Small number of rich documents, each embedding complex information
  - To group together properties typically accessed or written at the same time.
  - Information read or written in a single op.
  - Improved atomicity due to the simultaneous occurrence of mutations
  - Enhanced scalability due to fewer relations between objects.
  - Mutual consistency of grouped properties more easily maintained
- Large number of simple documents, each of which refers to others by keys
  - When access-patterns are predictable
  - When data-size needs to be kept small, in order to reduce network-bandwidth consumption.



```
key: order_200
{
  “orderID”: 200,
  “customer”:
    {“name”: “Steve Rothery”,
     “address”: “11-21 Paul Street”,
     “city”: “London”},
  “products”: [...]
}
```

```
key: order_200
{
  “orderID”: 200,
  “customer”:"customer_123",
  “products”: [...]
}
```

```
key: customer_123
{
  “name”: “Steve Rothery”,
  "email": "steveRothery@gmail.com"
   “address”: “11-21 Paul Street”,
   “city”: “London”
}
```

Embedded vs Referenced

## Simple mapping from RDBMS to Couchbase

Retain your data organization while enjoying a flexible JSON schema

| Relational Model | Couchbase |
|---|---|
| Server | Cluster |
| Database | Bucket |
| Schema | Scope |
| Table | Collection |
| Row | Document (JSON or BLOB) |
| Value | Sub-Document |

Note: Please note that Scope & Collections are available only from Couchbase Server 7.0

## Flexible Schema

Schema is the result of an application’s structuring of its documents

- Couchbase has flexible schemas and does not enforce rigid schemas.
  - In the document model, a schema is an arrangement of attribute-value pairs.
- Schemas are entirely defined and managed by applications.
- Document-structures can vary, even across multiple documents that each contain a type attribute with a common value.
- Differences between objects can be represented with great efficiency.
- Schema to be progressively evolved by an application, as required
  - Properties and structures can be added to a document, without other documents needing to be updated in the same way.

```
{
    "id": 102,
    "short.name": "JS",
    "%SS%": "091-55-1234",
    "name":{
       "first": "Jane",
       "last": "Smith"
    },
    "contact": {
        "@email": "js@gmail.com",
        "Office": {
          "cell#": "1-555-408-2345"
        }
    },
    "type":"customer"
}
```

```
{
    "short.name": "AD",
    "%SS%": "091-55-9876",
    "name":{
       "first": "Adam",
       "last": "Doe"
    },
    "type":"customer"
}
```

Schemas can vary

## Buckets

## Couchbase Server saves items in Buckets

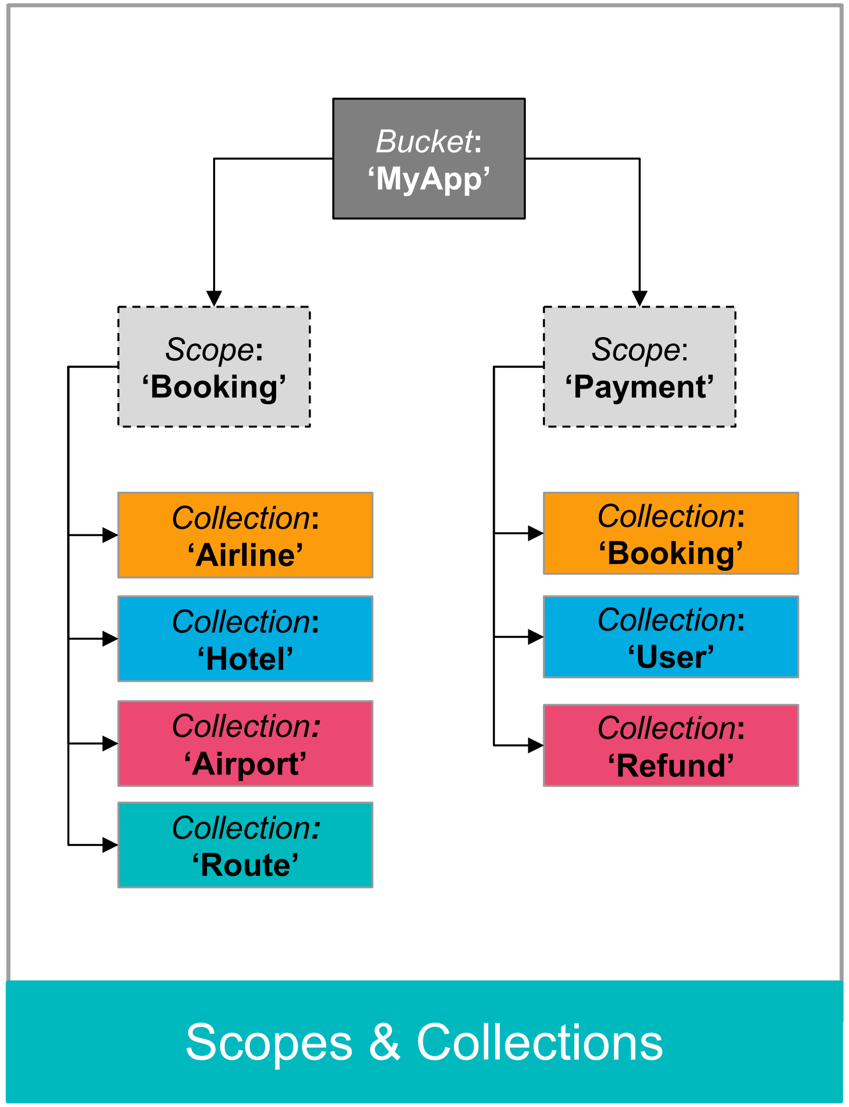
- Couchbase Server keeps items in **buckets**
- A maximum of 30 buckets can exist within a single cluster
- Each bucket has active and replica data sets (1-3 extra copies)
- A bucket has a type: *couchbase* bucket or *ephemeral* bucket

| Feature | Couchbase bucket | Ephemeral bucket |
|---|---|---|
| Bucket memory quota (per node) | Min 256MB | Min 256MB |
| Max Object Size | 20MB | 20MB |
| Persistence | yes | no |
| Replication and XDCR | yes | yes |
| Encrypted data access | yes | yes |
| Rebalance | yes | yes |
| N1QL, Seach, Analytics, Eventing | yes | yes |
| Indexing | yes | yes |
| Backup | yes | yes |

## Scopes and Collections

### Scopes and collections allow isolation/separation of data

- This allows to logical group together documents that look similar.
- A **collection** is a logical data container in a scope in a bucket
  - Item keys must be unique within their collection.
  - Items can optionally be assigned to different collections according to content-type.
  - Applications can be assigned per-collection access-rights.
- A **scope** is a mechanism for the logical grouping of multiple collections.
  - Collection-names must be unique within their scope.
  - Applications can be assigned per-scope access-rights.


*Bucket*: **'MyApp'**
*Scope*: **'Booking'**
*Scope*: **'Payment'**
*Collection*: **'Airline'**
*Collection*: **'Hotel'**
*Collection:* **'Airport'**
*Collection:* **'Route'**
*Collection*: **'Booking'**
*Collection*: **'User'**
*Collection*: **'Refund'**
Scopes & Collections

## Scopes and Collections - Benefits

Simplify ops and queries, increase efficiency of indexing, simplify migration from RDBMS and provide finer grain security

Logical grouping of similar documents potentially simplifies operations such as query, XDCR, and backup and restore.

Increased efficiency of indexing, due to the Data Service being able to provide documents from specific collections to the Index Service.

Simplified querying, since query statements are able to easily specify particular subsets of documents.

Easier migration from relational databases to Couchbase Server, since collections can be designed to correspond to pre-existing tables.

Secure isolation of different document-types, within a bucket; allowing apps to be authorized to use only their appropriate subsets of data

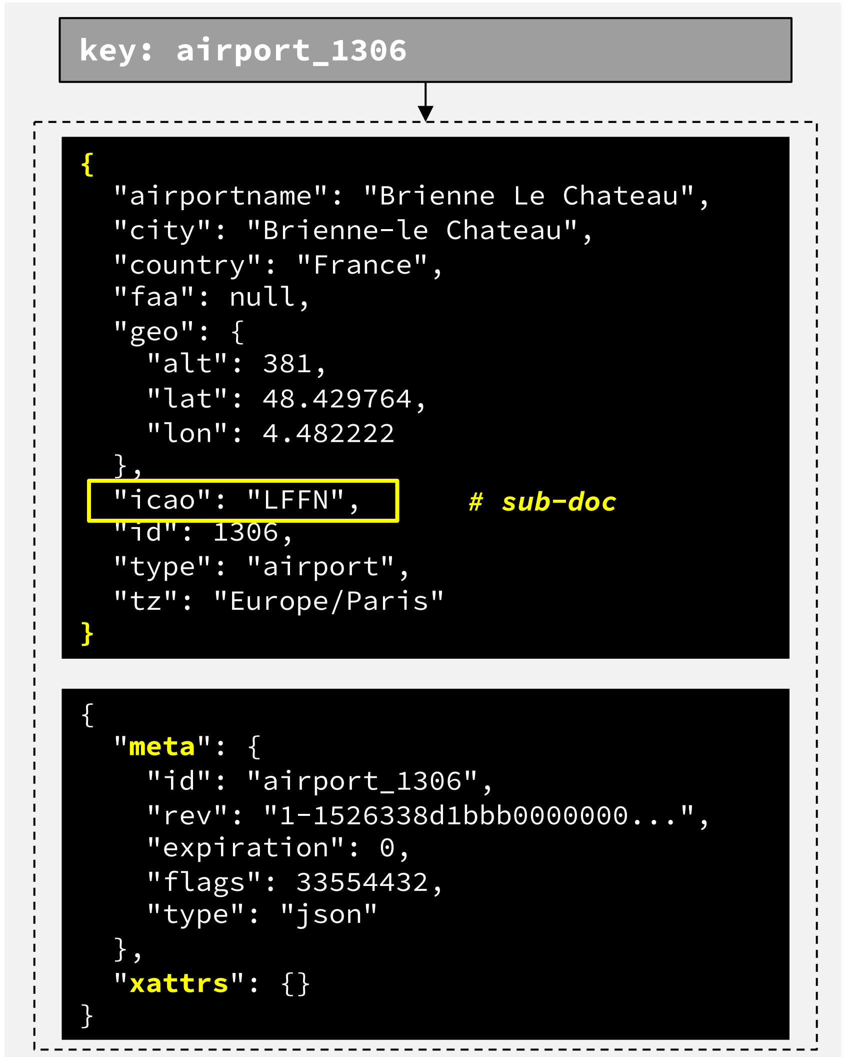
## Scopes and Collections Characteristics

| | Scope | Collection |
|---|---|---|
| Maximum number per cluster | 1000 | 1000 |
| Means of creation and management | • Couchbase Web Console UI<br>• Couchbase CLI<br>• Couchbase REST API<br>• N1QL<br>• Couchbase SDK | |
| User defined Naming convention | • Between 1 and 30 characters<br>• Can only contain the characters A-Z, a-z, 0-9, and the symbols _, -, and %.<br>• Case sensitive<br>• Cannot start with _ or % | |
| Indexable | No | Yes |
| Droppable | Yes. However, the default scope cannot be dropped | Yes. Default collection cannot be recreated once dropped |
| Replication with XDCR | Yes | Yes |
| Access | Protected by Role-Based Access Control (RBAC); appropriate roles must be assigned | |

## Data and Metadata

Couchbase Server saves data as items, each of which has a key and a value.

- Each item consists of a **key** and a **value**.
  - Each key must be unique within a collection
  - Each key may be no longer than 246 bytes.
  - Values can be either *binary* or *JSON* documents.
  - The maximum size of a value is 20 MiB.
- A **sub-document** is an inner component of a JSON document
  - The sub-doc is referred to by a path
  - The Couchbase SDK provides a Sub-Document API
- **Metadata** is automatically generated and stored for each item saved in Couchbase Server
  - meta: id, rev, expiration, flags, type
  - xattrs: special kind of metadata



```
key: airport_1306
```

```
{
  "airportname": "Brienne Le Chateau",
  "city": "Brienne-le Chateau",
  "country": "France",
  "faa": null,
  "geo": {
    "alt": 381,
    "lat": 48.429764,
    "lon": 4.482222
  },
  "icao": "LFFN",          # sub-doc
  "id": 1306,
  "type": "airport",
  "tz": "Europe/Paris"
}
```

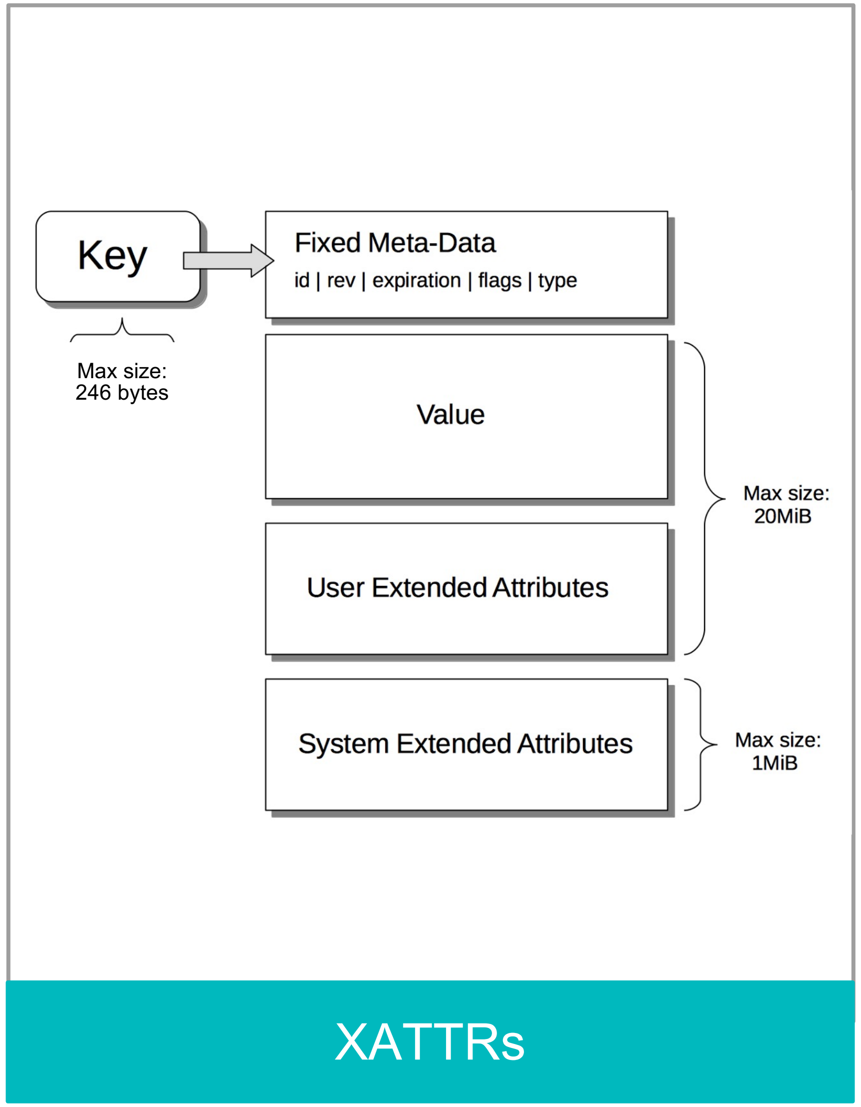
```
{
  "meta": {
    "id": "airport_1306",
    "rev": "1-1526338d1bbb0000000...",
    "expiration": 0,
    "flags": 33554432,
    "type": "json"
  },
  "xattrs": {}
}
```

Key, Value and Metadata

## Extended Attributes

Couchbase Server permits the definition of extended attributes (XATTRs)

- **User XATTRs** allow developers to define application-specific metadata, for use in libraries and frameworks
  - The application can access and modify the User XATTR it has created within a document.
  - This is achieved by extensions to the Sub-Document API
  - User XATTR can be formed only as JSON
- User Extended attributes count against the maximum size of 20 megabytes for each document
- **System XATTRS** are Server-defined
  - Used for Mobile support
  - Used for Transaction support



# Indexes

## Indexes enhance the performance of query and search operations.

- Indexes are used by certain services, such as Query, Analytics, and Search, as targets for search-routines.
- Indexes, when well-designed, provide significant enhancements to the performance of search-operations.
- The Query service relies on indexes provided by the Index service.
- The Search and Analytics services both provide their own indexes, internally.

| Type | Service | Description |
|---|---|---|
| **Primary** | Index | Based on key of item in a collection. Maintained asynchronously. To be used for simple queries. |
| **Secondary** | Index | Based on an attribute within a document. The value associated with the attribute can be of any type: scalar, object, or array. Used most frequently for N1QL queries |
| **Full Text** | Search | Derived from textual contents of documents within one or more keyspaces. Text-matches of different degrees of exactitude. Can be purged of irrelevant chars. |
| **Analytics** | Analytics | Materialized access for shadow data in an Analytics collection to speed up Analytics queries. Updated automatically. |
| **View** | Data | Deprecated. |

Index types

## Durability

Durability ensures the greatest likelihood of data-writes surviving unexpected anomalies, such as node-outages.

- By default, all replication and persistence operations are asynchronous and eventual.
- A client may request stricter consistency and/or durability on any request by specifying a *Durability Level*.

| Durability Level | Description |
|---|---|
| NONE | (Default) Write data to memory of active data node before returning |
| MAJORITY | Write data to a majority of data nodes with copies of this vbucket |
| MAJORITY_AND_PERSIST_TO_ACTIVE | Same as MAJORITY plus persist to disk on active node |
| PERSIST_TO_MAJORITY | Same as MAJORITY plus persist to disk on majority of data nodes |

```
Collection collection = bucket.defaultCollection();
JsonObject doc = JsonObject.create().put("myElement","new value");
// Ensure that write occurs to a majority of replicas
collection.upsert(id, doc, upsertOptions().durability(DurabilityLevel.MAJORITY) );
```

## Transactions

Couchbase Server supports Transactions across multiple documents in more than one shards on different nodes

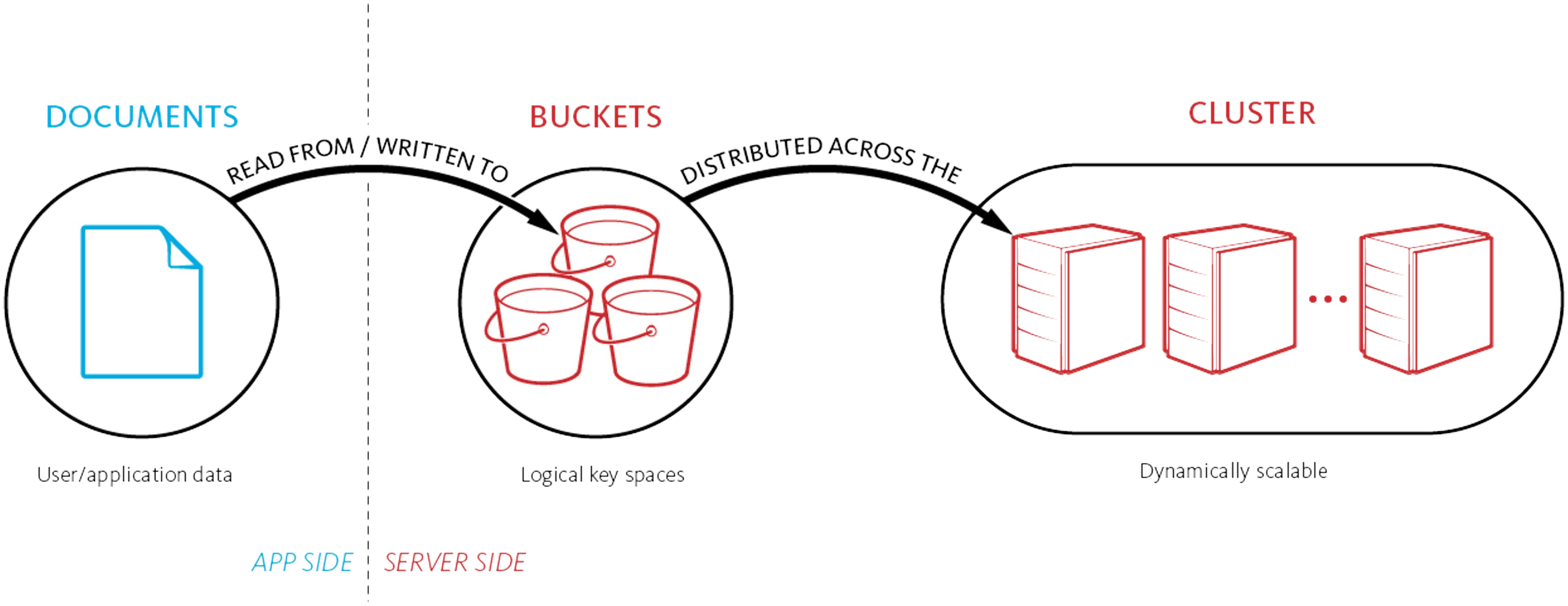
- A transaction is an atomic unit of work that contains one or more operations either committed to the database together or all undone.
- Transaction APIs supports **Key-Value** insert, update, and delete operations, across any number of documents.
- Transaction APIs also supports a seamless **integration between Key-Value and Query DML** statements (SELECT, INSERT, UPDATE, DELETE, UPSERT, MERGE) within transactions.
- **Multiple** Key-Value and Query DML statements can be used together inside a transaction.

```
// Bulk update salaries of all employees
// in a department
transactions.run((ctx) -> {
  var auditLine =
        JsonObject.create()
                   .put("content",
                        "Update on 4/20/2020");
             ctx.insert(auditCollection,
                  "Dept10",
                   auditLine);
             ctx.query("UPDATE employees
                              SET salary = salary
* 1.1
                               WHERE dept = 10
                   AND salary < 50000");
             ctx.query("UPDATE department
                              SET status =
'updated'
                              WHERE dept = 10");
             txnctx.commit();

  });
```

Transaction example

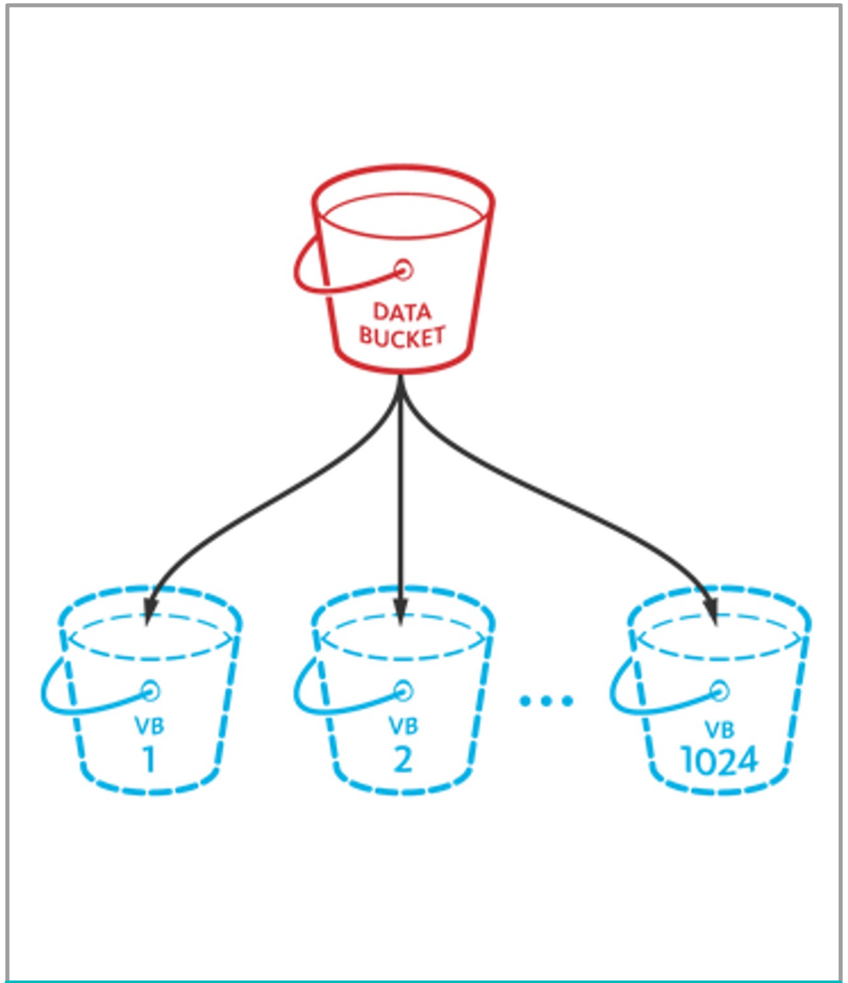
## Buckets


DOCUMENTS
READ FROM / WRITTEN TO
BUCKETS
DISTRIBUTED ACROSS THE
CLUSTER
User/application data
Logical key spaces
Dynamically scalable
APP SIDE
SERVER SIDE

## Virtual Buckets

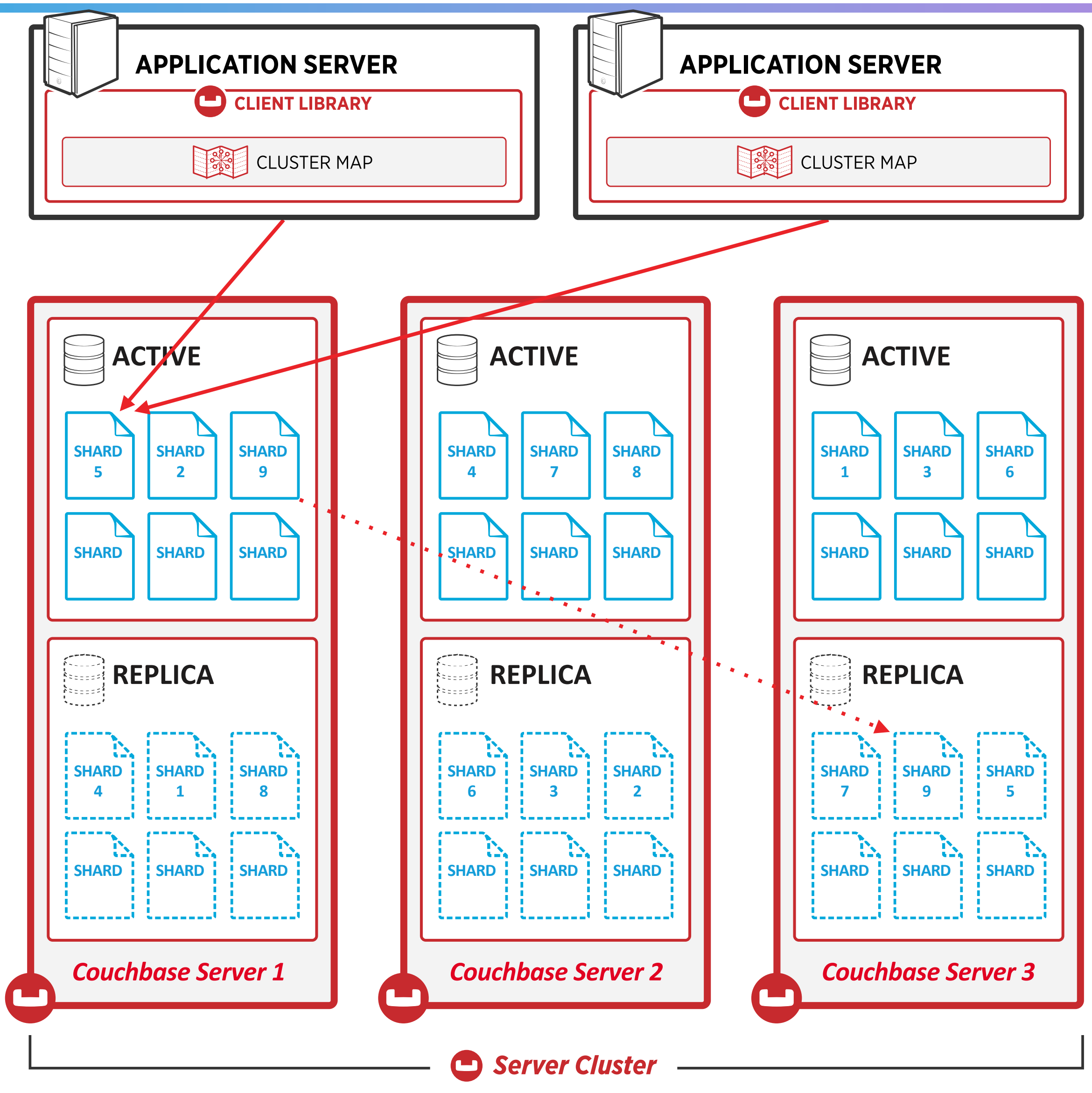
*vBuckets* help distribute data effectively across a cluster

- Each bucket is split into 1024 Virtual Buckets or vBuckets
  - vBuckets are sometimes called *shards*
- Each vBucket contains 1/1024th portion of the data set.
- vBuckets do not have a fixed physical server location. They are mapped to individual nodes by the Cluster Manager.
- Multiple vBuckets are stored within the same physical node.
- vBuckets can be moved between nodes if the cluster topology changes.



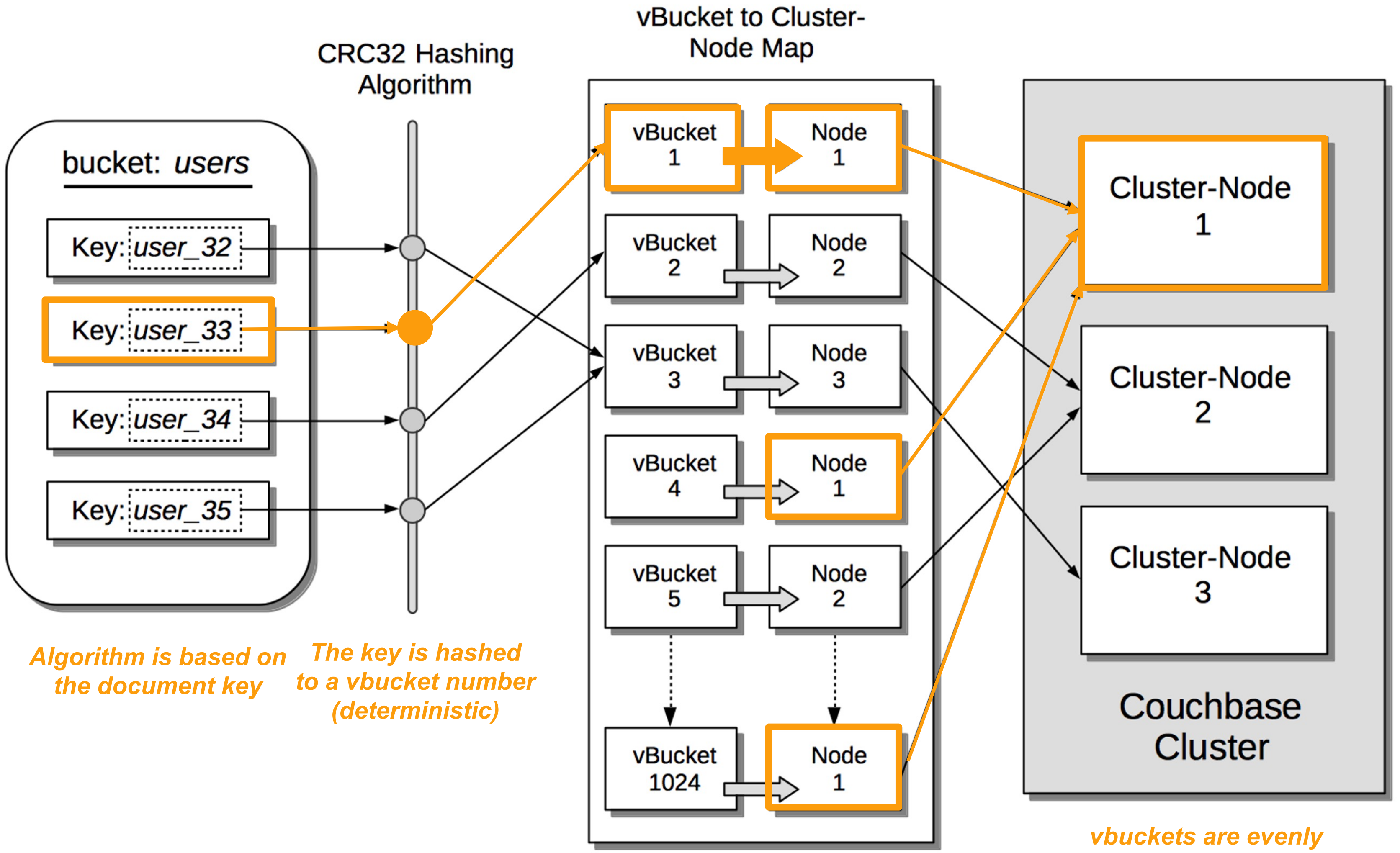
virtual Buckets

# Sharding Architecture



## Auto-Sharding

Document keys always get hashed to the same vBucket using a CRC32 hashing function



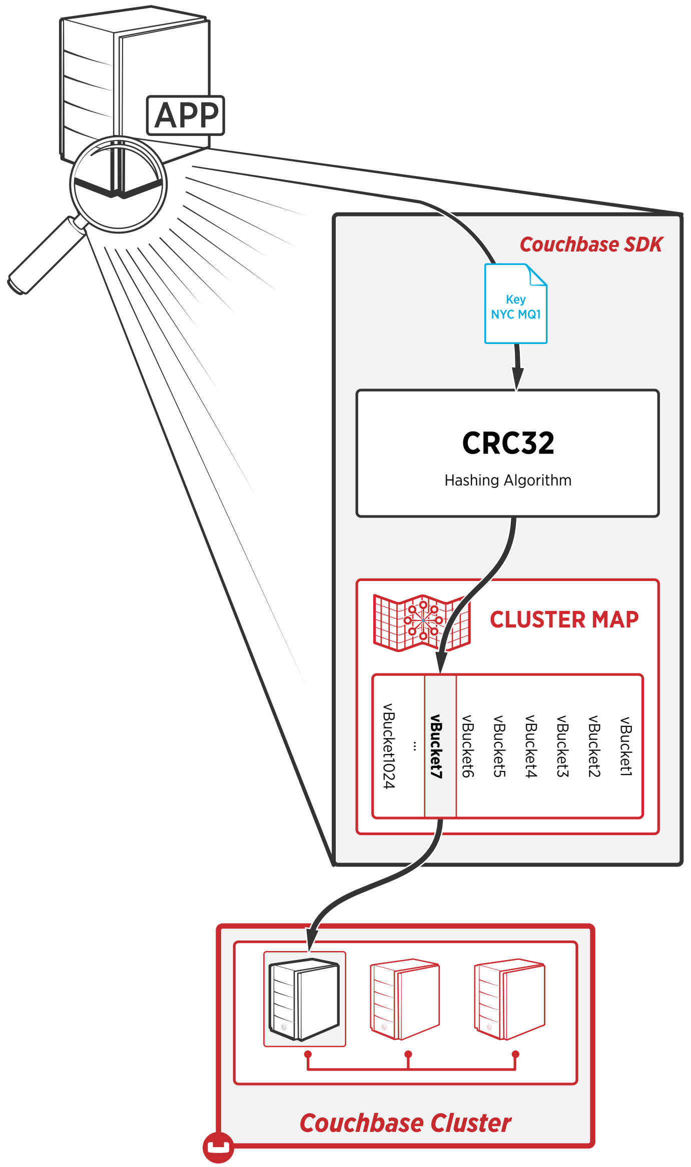
*Algorithm is based on the document key*

*The key is hashed to a vbucket number (deterministic)*

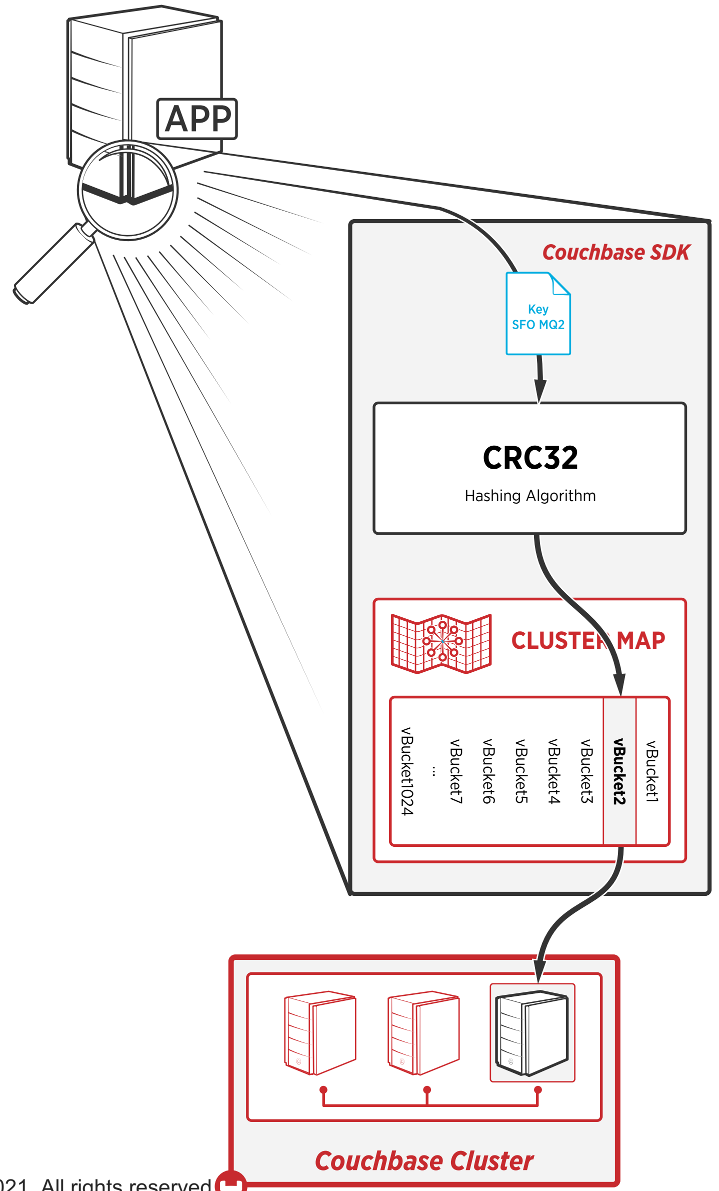
*The vbucket number is mapped to a node (dynamic)*

*vbuckets are evenly distributed*

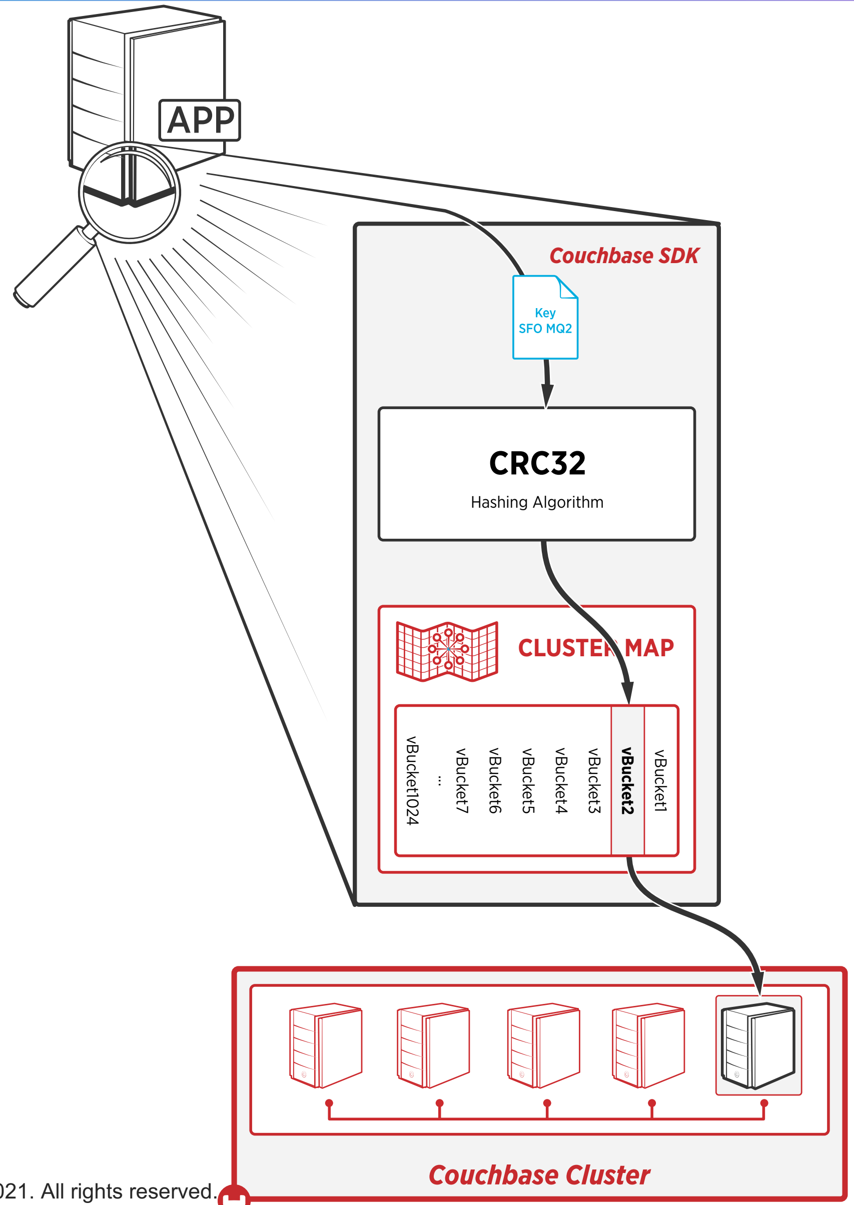
## Cluster Map


APP
Couchbase SDK
Key
NYC MQ1
CRC32
Hashing Algorithm
CLUSTER MAP
vBucket1
vBucket2
vBucket3
vBucket4
vBucket5
vBucket6
vBucket7
...
vBucket1024
Couchbase Cluster

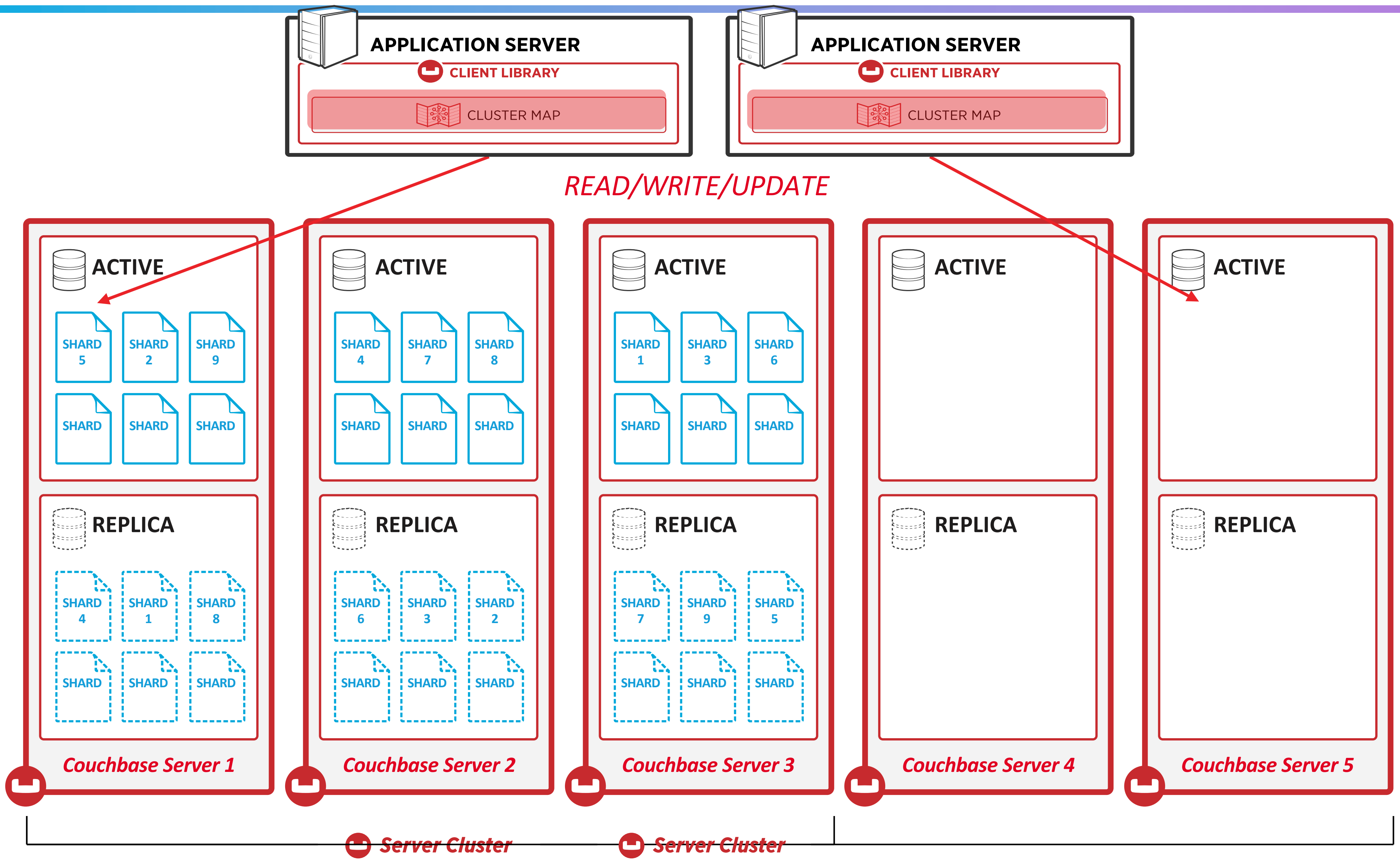
## Cluster Map


APP
Couchbase SDK
Key
SFO MQ2
CRC32
Hashing Algorithm
CLUSTER MAP
vBucket1
vBucket2
vBucket3
vBucket4
vBucket5
vBucket6
vBucket7
...
vBucket1024
Couchbase Cluster

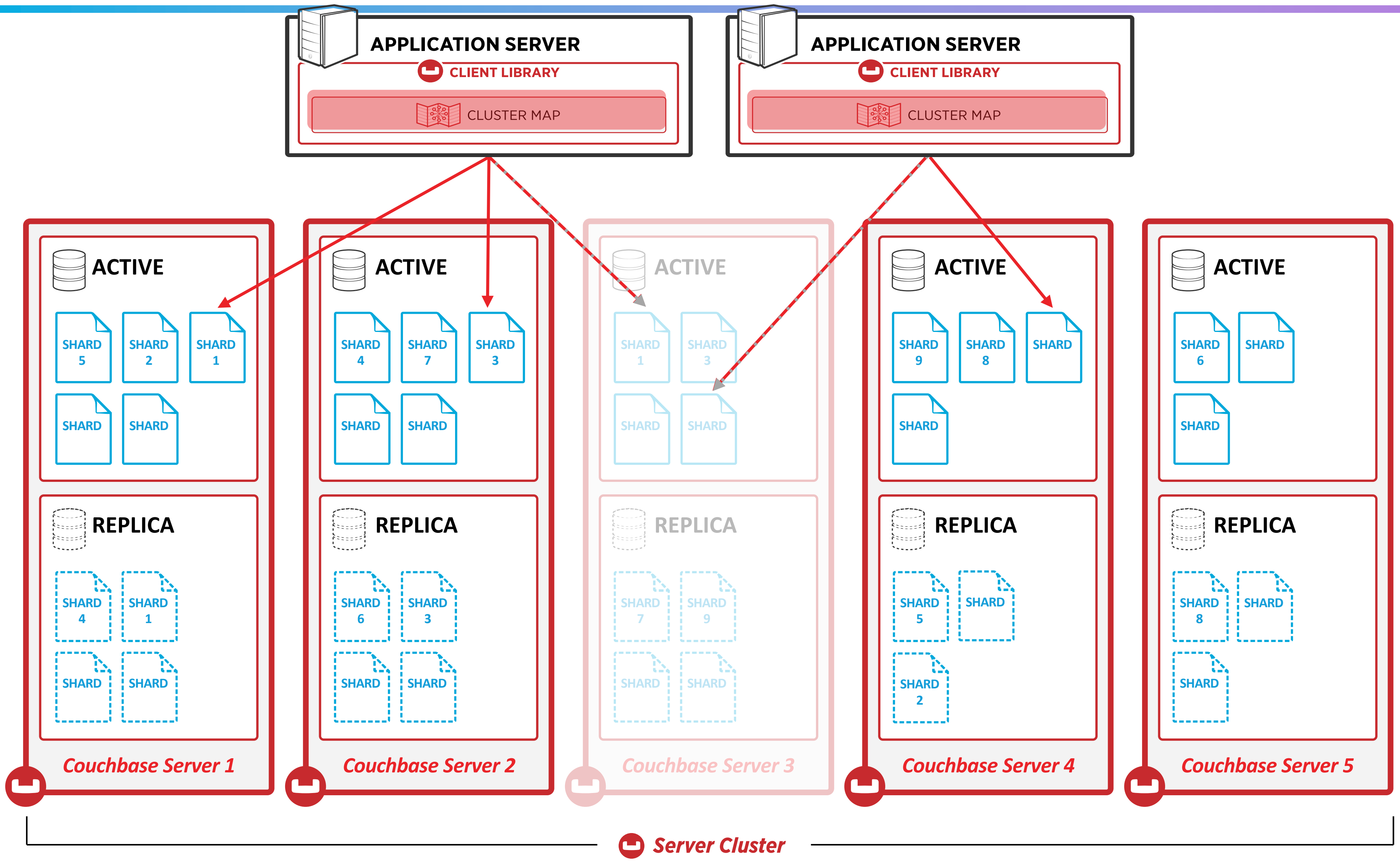
# Cluster Map – Addition of 2 Nodes



# Adding Nodes to Cluster



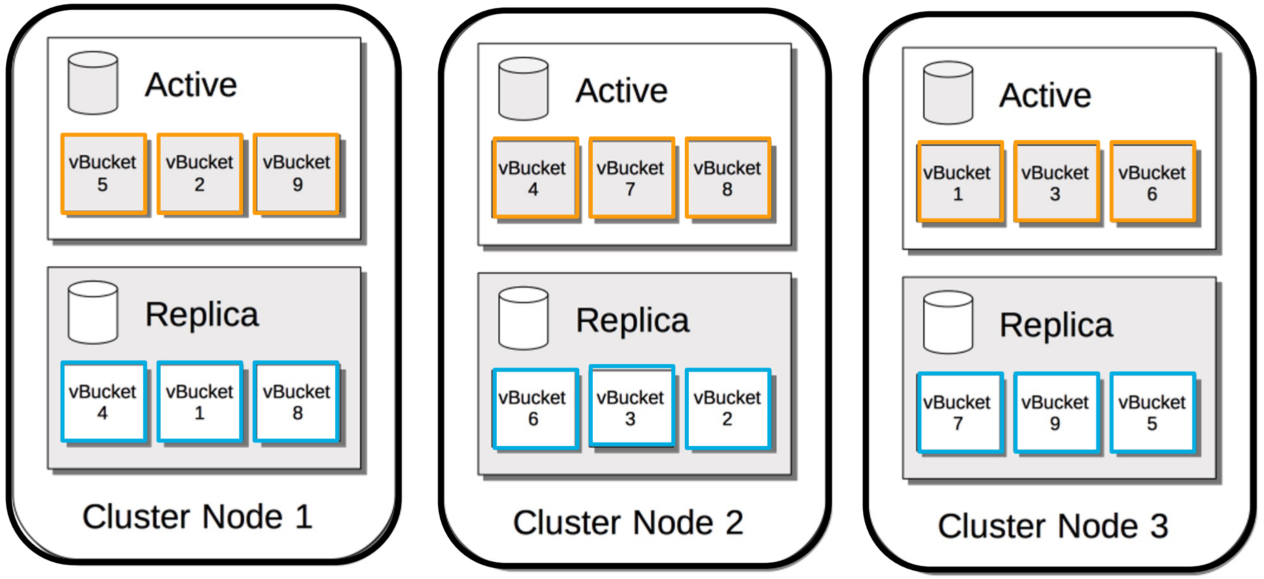
# Node Failover



## Replica

Intra-cluster replication involves replicating data across the nodes of a cluster.

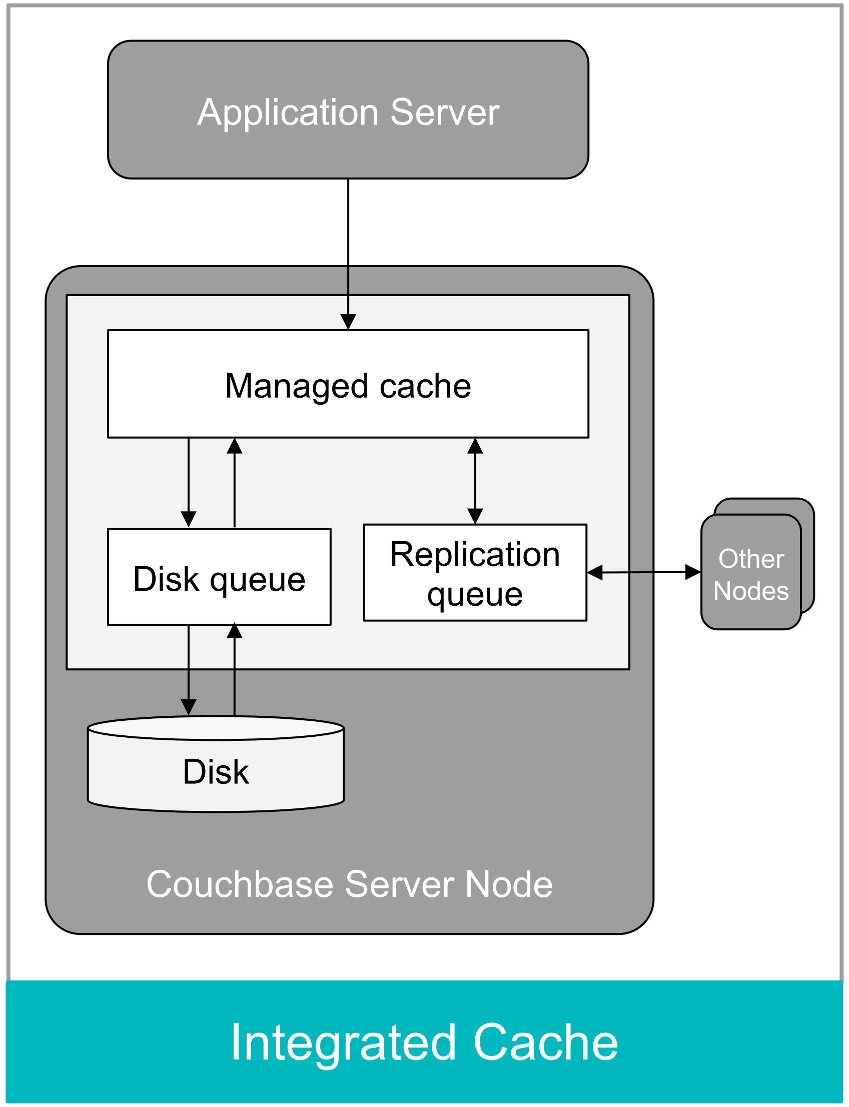
- Up to 3 replica buckets can be defined for every bucket.
- Each replica itself is also implemented as 1024 vBuckets.
- One of the vBucket copies is *Active* at any point and services read and write request. The *Active* vBucket will change if a node fails over.
- *Replica* vBuckets receive a continuous stream of mutations from the *Active* vBucket and are thereby kept constantly up to date.



## Managed Cache

Couchbase Server provides a fully integrated caching layer, which provides high-speed data-access

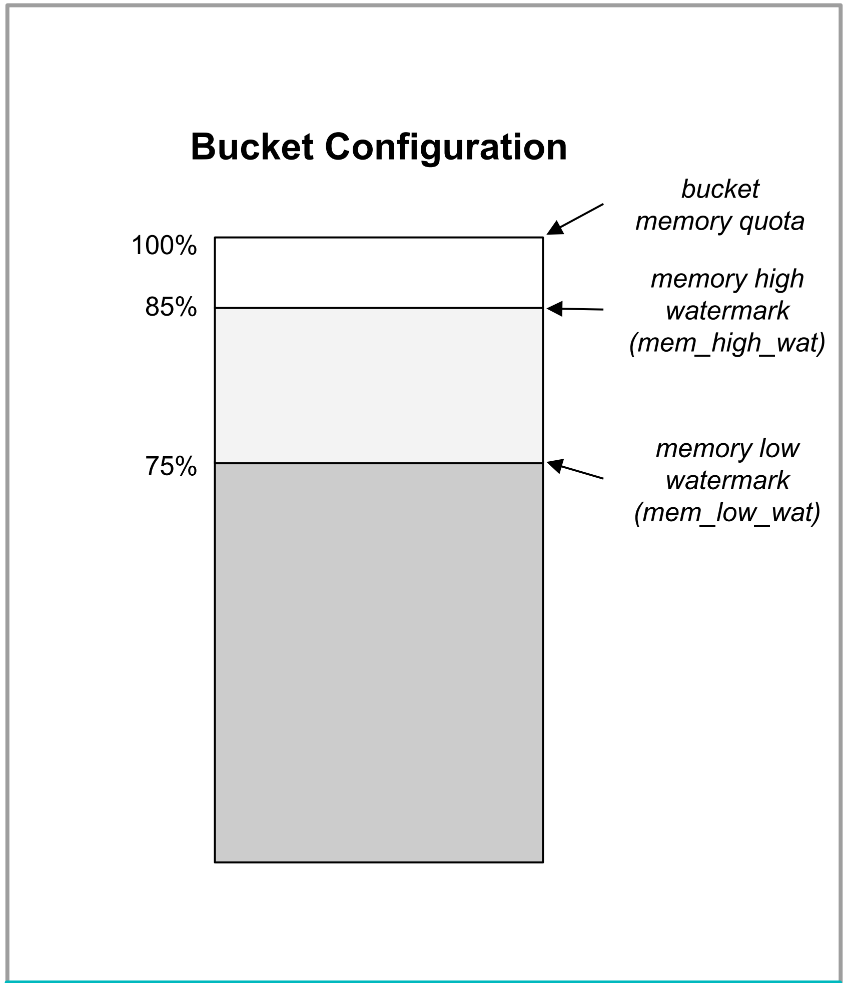
- Applications write and read data from the managed cache
- Couchbase Server automatically manages the cache
  - Items that have been newly created or updated are placed on a replication queue, so that one or more replica buckets can be created or updated.
  - They are also placed on the disk queue to be written to disk.
- Once written, a new document resides both in the memory and on the disk of the node.
  - By default, the app gets the write callback when items are in the cache.
- Note: both active items and replicated items are stored in the managed cache



## Cache Ejection

### Cache ejection occurs when the cache is almost full

- If a bucket's memory quota is almost reached, items are ejected from the cache
- Item is removed from memory, but not from disk
- For each bucket, when the fill rate of the cache hits the `mem_high_wat`, items are ejected from the cache until the quantity of data has decreased to the `mem_low_wat`
- If ejection cannot free enough space to support continued data-ingestion, the server will stop ingesting data and send errors. When sufficient memory is again available, data-ingestion resumes.


Bucket Configuration
100%
85%
75%
bucket memory quota
memory high watermark (mem_high_wat)
memory low watermark (mem_low_wat)

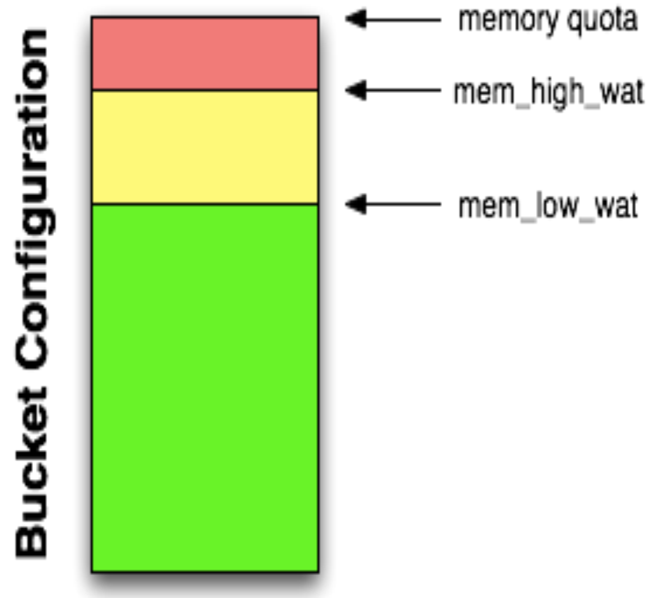
Memory management

## Cache Management

When memory usage passes:

- Low watermark:
    - Nothing
- High watermark: NRU used first, random afterwards
    - Ejection continues as needed until low watermark reached
- Overhead is built-in to leave space for:
    - unexpected spikes
    - rebalancing

Best practice is to keep “working set” (some percentage of overall dataset) below high watermark for best performance



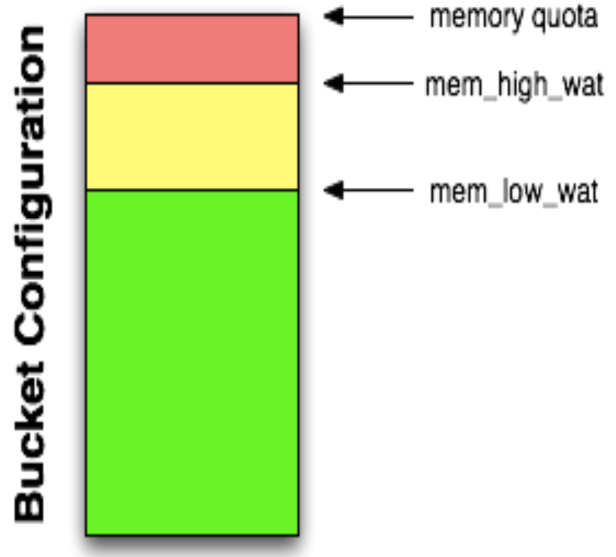
## Out-of-Memory Errors

If memory usage reaches 90% of quota, clients are told to temporarily back off (“temp_oom” )

- Typically caused by too many/spike in writes

If memory cannot be reclaimed, clients are told system is full (“oom”)

- Typically caused by RAM being filled with metadata



## Working Set Management

The high water mark and low water mark values can be tuned for performance

- 85% high water mark default
- 75% low water mark default
- Configured as percentage or absolute RAM usage

Balance of replica to active ejection after NRU can be set

- 60/40 replica to active by default

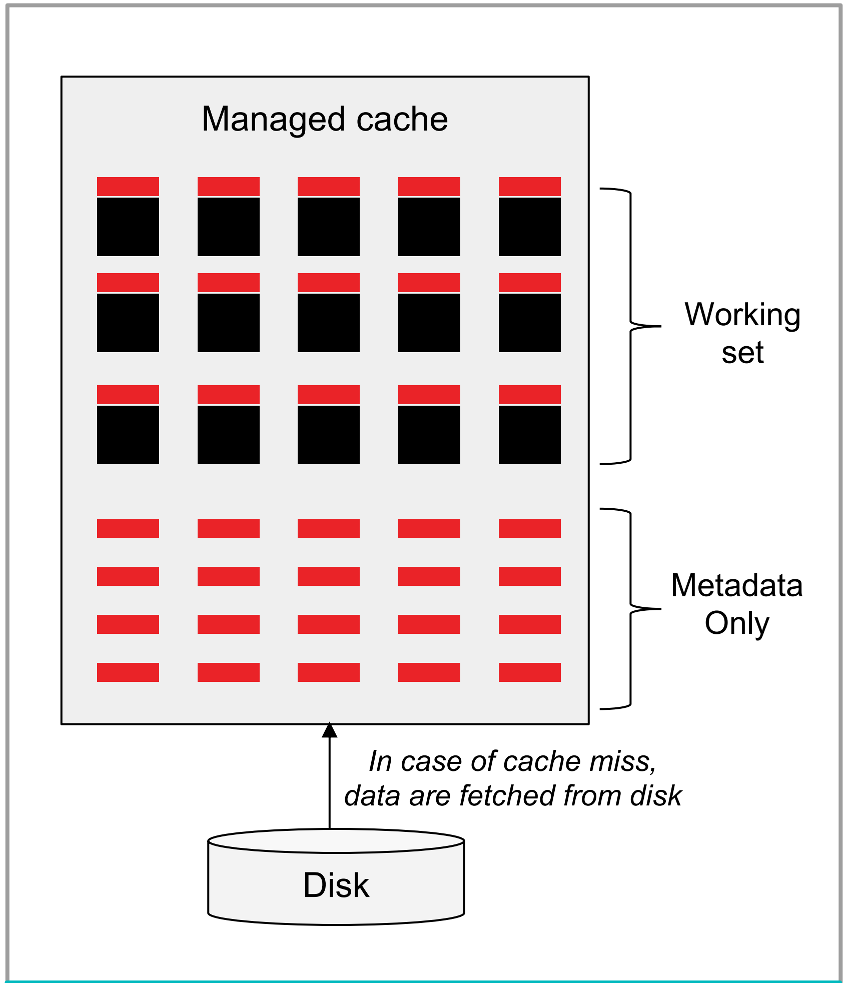
Expiry pager interval can be changed

- 60 minutes by default

## Cache Ejection Policies

Different ejection methods are available, and are configured per bucket

- The node uses the metadata in RAM to quickly determine:
  - If an item is missing so it can add it in the cache
  - Which items less recently used are candidate for ejection
- Ejection policy determines if metadata are ejected or not.
- **Value-only**: Only items are removed. Metadata stay in memory. This favors performance at the expense of memory.
  - Note: make sure the working set is high enough to avoid perfs. issues
- **Full**: Both items and metadatas are removed. This favors lower memory consumption versus performance.

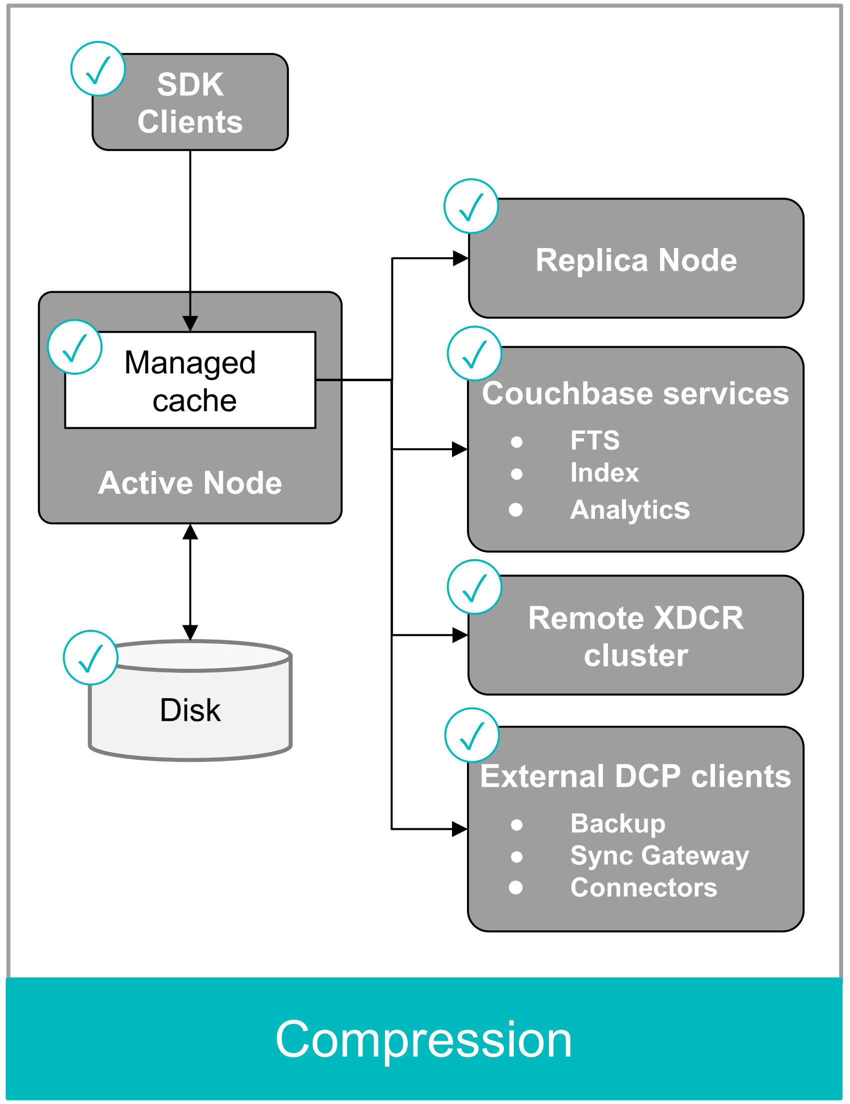


Example of Value-only ejection

# Compression

Couchbase Server supports data compression in its communications with internal and external clients, and in its internal handling of items.

- The compression allows *RAM* and *disk* to be used with increased efficiency. It may also reduce consumption of *network bandwidth*.
- Compression applies to both binary and JSON items.
- Couchbase SDK clients, Replica nodes, internal Couchbase Services, remote XDCR clusters and external DCP clients can use compression.
- Couchbase Server may (depending on the mode of the bucket) store items in compressed form in memory.
- The server always compresses items when storing them on disk.



# Compression Modes

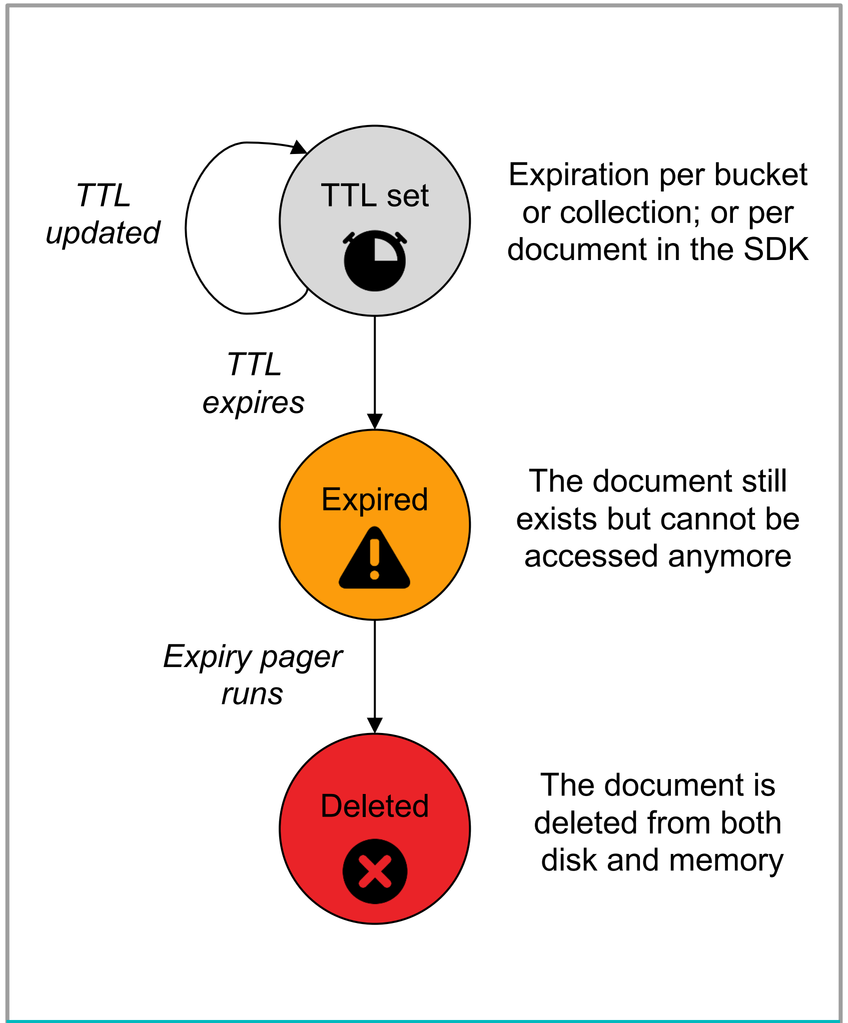
The Couchbase server runs compression and decompression depending on the mode supported by the bucket and by the client

| Mode | Use Case | Description |
|---|---|---|
| Off | Recommended where clients cannot benefit from compression, and where neither memory-resources nor network-bandwidth will be negatively impacted by the size of items. | ● On receipt of a compressed item, Couchbase Server decompresses the item when storing in memory; and recompress it when storing on disk.<br>● Couchbase Server sends the item uncompressed. |
| Passive | **This mode is assigned by default.**<br>It supports clients that handle compression themselves.<br>It allows Couchbase Server to limit its use of memory resources and network.<br>It does not force Couchbase Server to use CPU resources for the compression and decompression of items that clients do not themselves require in compressed form. | ● On receipt of a compressed item, Couchbase Server stores it compressed both in memory and on disk. It sends the compressed item back to the client if this is requested by the client; otherwise, uncompressed.<br>● On receipt of an uncompressed item, Couchbase Server stores it uncompressed in memory, and stores it on disk compressed. It returns the item to the client uncompressed. |
| Active | This mode allows the server to practice the maximum conservation of memory resources and network bandwidth.<br>More items can be held in memory and accessed with improved overall efficiency.<br>In some circumstances, clients that do not themselves require compression may be negatively affected. | ● Couchbase Server actively compresses items for storage in memory and on disk, even if the items are received uncompressed.<br>● Items are decompressed before being sent back to clients that do not support compressed data.<br>● Items are sent compressed to clients that do support compressed data, even if those clients originally sent the items to the server in uncompressed form. |

## Expiration

Time To Live (TTL) imposes a maximum lifespan on items to ensure the expiration of such items, once the specified period is complete.

- TTL is specified per bucket, per collection or per document
  - The default value is 0, which indicates that TTL is disabled.
- If TTL is enabled, each newly created item lives for duration specified by TTL.
  - TTL can also be updated before it expires
- After its expiration time is reached, the item will be deleted by Couchbase Server.
  - Deleted items and their contents are no longer available: access-attempts are treated as if the items never existed.
  - A process *Expiry Pager* scans for items that have expired every 10 mins and erases them from memory and disk.



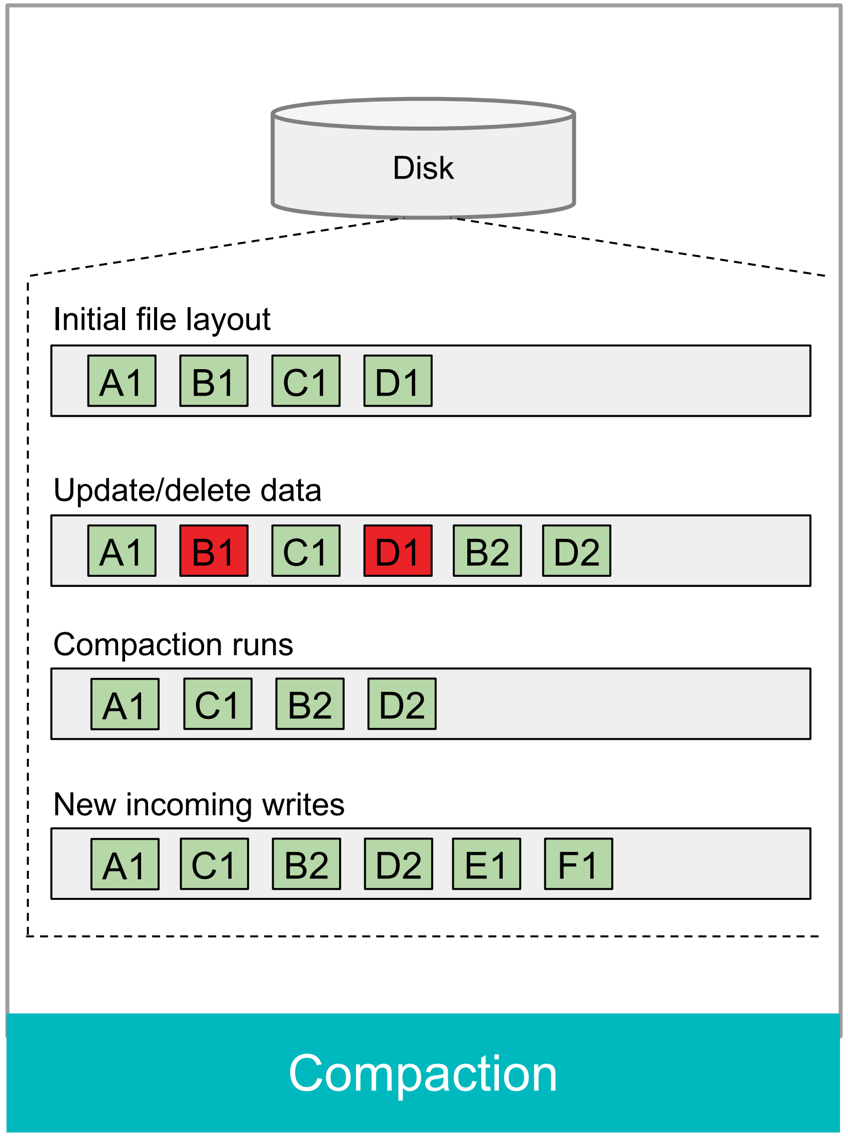
TTL updated
TTL set
Expiration per bucket or collection; or per document in the SDK
TTL expires
Expired
The document still exists but cannot be accessed anymore
Expiry pager runs
Deleted
The document is deleted from both disk and memory

Lifecycle of document with TTL

## Storage

Couchbase Server provides persistence, whereby all items are eventually persisted to disk; and durability is thereby enhanced.

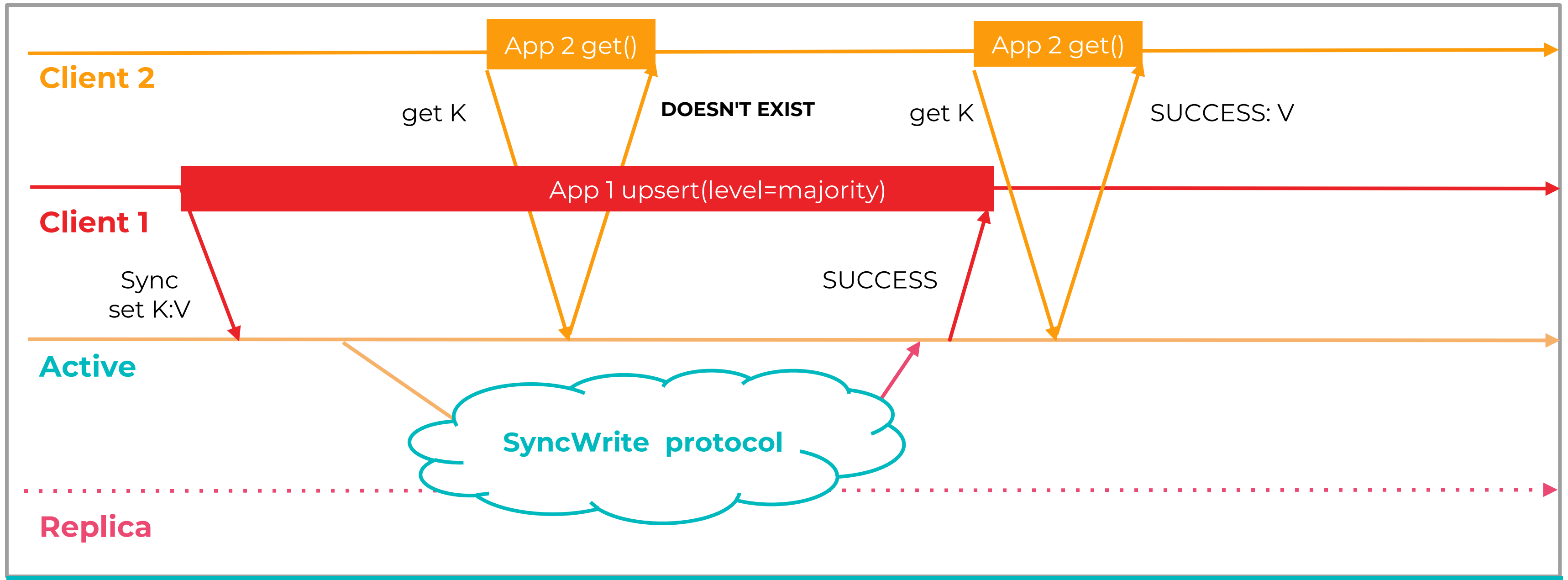
- Disk paths
  - Node specific; up to 4 custom paths.
- Threading
  - Multi-threaded readers and writers provide high-performance operations for data on disk; Min 4, Max 64 per node.
- Tombstones
  - A record of a deleted item is maintained to provide eventual consistency between nodes and clusters; Tombstones are removed during compaction.
- Compaction
  - Couchbase Server uses an append-only file-write format. File-sizes are reduced periodically by compaction.
- Disk I/O priority
  - One bucket's disk I/O can be granted priority over another's.



## Durable Writes

A durable write is synchronously replicated and/or persisted

- Durable operation only responds to client (return success / failure) once it is *replicated and/or persisted based on the 3 levels of tunable durability*.
- Other clients reading the document *before* write is durable will return the old (pre-write) value.



Atomicity of Durable Writes