

# Couchbase Database Service

## List of Contents

- Introduction
- Key Concepts
- Key Terminologies
- Architecture
- Installation and Configuration
- CRUD Example Using CBQ
- CRUD Example Using Spring Boot Application

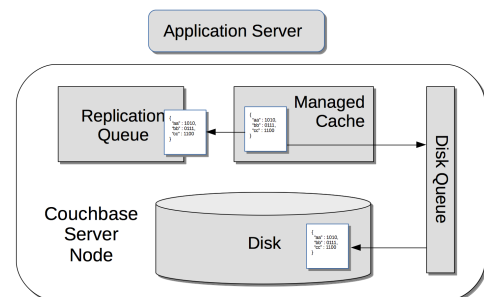
## Introduction

Couchbase is a NoSQL document-oriented database management system. It makes use of a key-value structure to provide flexible and effective data management. Additionally, it has functions like full-text search, querying, and indexing. Numerous use cases, like online gaming, e-commerce, and mobile apps, are possible with Couchbase. Let's discuss few of its key characteristics:

## Key Concepts

### Memory First Architecture

Couchbase's memory-first architecture uses a combination of in-memory data storage and disk-based persistence to provide high performance and low latency for data access. This architecture allows the database to store and retrieve data directly from memory, rather than reading from disk, which greatly reduces the time required for data access. Additionally, data is asynchronously and continuously persisted to disk, providing durability in the event of a system failure. This allows for a balance of high performance and data reliability.

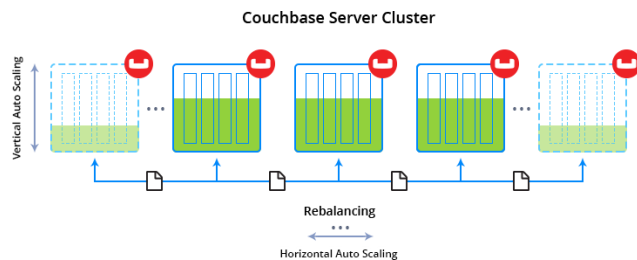


### Cluster

In a Couchbase cluster, data is stored in RAM across multiple nodes. The cluster uses a distributed data architecture, which allows for data to be partitioned and distributed evenly across the nodes in the cluster. Each node in the cluster has its own memory, and data is stored in the memory of the node that is responsible for managing that particular piece of data.

## Fault Tolerant and High Availability

Couchbase provides fault tolerance and high availability through its distributed data architecture and automatic failover mechanisms. It replicates data across multiple



nodes in a cluster, so that if a node fails, the data can still be accessed from another node. Additionally, it automatically detects and recovers from node failures, minimizing downtime and ensuring that data is always available.

## Memory Eviction

When the RAM of a node becomes full, Couchbase uses a process called "memory eviction" to free up memory. This process works by removing the least recently used (LRU) items from memory and storing them on disk. This helps to maintain a balance between performance and storage capacity and ensures that the most frequently accessed data remains in memory for fast access. Couchbase also uses various eviction policies to determine which items to evict when memory is full.

## Key Value Pair

Couchbase uses a key-value data model, where each piece of data is stored as a unique key-value pair, with the key being a unique identifier and the value being the actual data. This model allows for fast and efficient data access, making it well-suited for high-volume read and write use cases. Additionally, it supports various data types including JSON, BLOB, and more, which allows for flexible data modeling.

## Key Concepts

### N1QL

N1QL (Non-first normal form Query Language) is a SQL-like query language for Couchbase. It allows developers to query and manipulate JSON data using familiar SQL syntax, making it easy to work with and understand. N1QL also provides indexing, joins and aggregation functionality, enabling advanced querying capabilities for Couchbase.

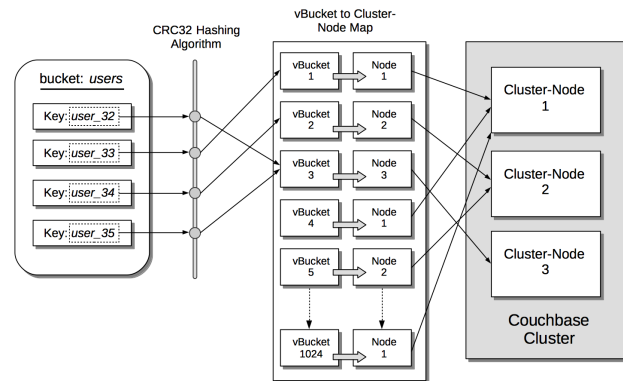
### Bucket

A bucket in Couchbase is a container for data, similar to a table in a relational database. Each bucket stores a specific set of documents, and is used to organize and manage

data within a Couchbase cluster. Buckets can be created, deleted and modified, and they also can have their own configuration settings such as memory allocation and eviction policies.

## **VBucket**

A vBucket (virtual bucket) is a logical partition of a bucket. Couchbase uses a distributed data architecture, so data is partitioned and distributed across multiple nodes in a cluster. vBuckets are used to manage this partitioning, allowing data to be evenly distributed and ensuring that each node in the cluster has a consistent and manageable amount of data. Each vBucket is responsible for a specific range of keys and is associated with a specific node, this way data is evenly distributed and easily manageable. The default number of vBuckets for a bucket is 1024. However, this number can be adjusted during the creation of a bucket. The number of vBuckets is determined by the number of nodes in the cluster and the amount of data that needs to be stored.



## **Compression**

Couchbase supports data compression to reduce the amount of storage space required for data and to improve network transfer performance. Compression can be enabled or disabled on a per-bucket basis, allowing for flexible data management. When data is stored in memory, it may be stored in an uncompressed format for faster access, however when it is evicted from memory, it is stored on disk in the format (compressed or uncompressed) that is specified by the bucket configuration.

## **Indexing**

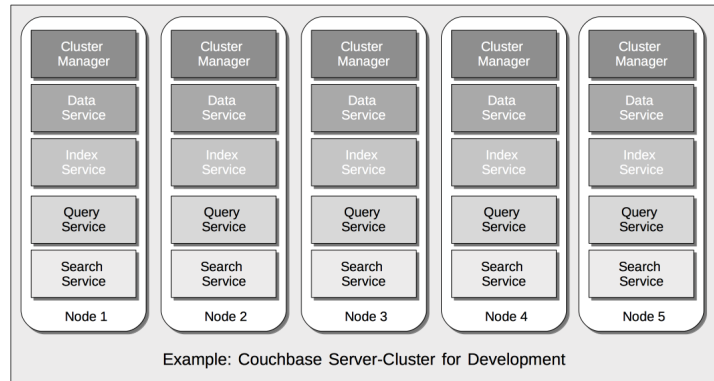
Couchbase provides indexing through GSI, which allows for the creation of indexes on data stored in a bucket to optimize query performance.

- Primary indexing is the default indexing mechanism, it creates an index on the primary key of the document and it is always present, it cannot be removed.
- Secondary indexing (GSI) allows for the creation of additional indexes on specific fields within the JSON documents. This allows for querying and filtering based on the value of specific attributes or fields.
- Composite indexing is a type of secondary indexing where multiple fields can be indexed together, this allows for querying multiple fields at once.

## Architecture

Couchbase has a distributed architecture that is designed for performance, scalability, and availability. It uses a shared-nothing architecture, where each node in the cluster is independent and self-sufficient. The main components of the Couchbase architecture are:

- **Data service:** These are the nodes that store and manage the data in the cluster. They handle data operations such as reads, writes, and queries.
- **Index service:** These nodes are responsible for creating and maintaining indexes on the data stored in the cluster. They handle indexing operations such as creating, updating, and querying indexes.
- **Query service:** These nodes handle query operations on the data stored in the cluster. They use the indexes created by the index nodes to execute queries and return results to the client.
- **Cluster Manager:** This component is responsible for managing the nodes in the cluster, including adding and removing nodes, monitoring the health of the cluster, and rebalancing data across the nodes.
- **Couchbase Server:** It is a distributed key-value store, document database, and caching system.



Couchbase also supports various features like Eventing, Analytics, and Search which can be integrated with the Couchbase server for advanced querying and processing capabilities.

## Installation and Configuration

1. Follow the official documentation for downloading and installing Couchbase Server: <https://docs.couchbase.com/server/current/install/install-intro.html>
2. On Windows, Couchbase Server is installed as a Windows service. By default, the Couchbase Server service automatically starts when the system boots. On other OS start the Couchbase Server service and open the Web Console in a web browser by navigating to <http://localhost:8091>
3. Create a new cluster by following the prompts in the Web Console.
4. Add nodes to the cluster by clicking on the "Add Node" button in the Web Console. You can enable/disable services while adding a node.
5. Configure the cluster settings, such as data and index memory, by navigating to the "Settings" tab in the Web Console.
6. Create a new bucket to store your data by navigating to the "Data Buckets" tab in the Web Console.
7. Test the Couchbase Server by inserting and retrieving data using the built-in SDKs or the Web Console.
8. We can use the Couchbase's Query Workbench to test the N1QL queries.

Servers

cluster

Dashboard

Servers

XDCR

Security

Settings

Logs

data

Buckets

Query

Indexes

Active Servers

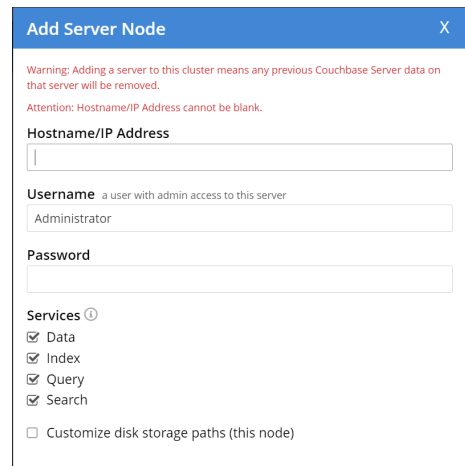
Pending Rebalance

Server Groups

+ Add Server

Rebalance

Server Node Name	Group	Services	RAM Usage	Swap Usage	CPU Usage	Data/Disk Usage	Active/Replica Items	
127.0.0.1	Group 1	<div>Data</div> <div>Full Text</div> <div>Index</div> <div>Query</div>	<div><div></div>36.4%</div>	<div><div></div>38.8%</div>	<div><div></div>29.6%</div>	4.04MB / 4.06MB	0 / 0	<div>Remove</div> <div>Failover</div>



**Add Server Node** X

Warning: Adding a server to this cluster means any previous Couchbase Server data on that server will be removed.

Attention: Hostname/IP Address cannot be blank.

Hostname/IP Address

Username a user with admin access to this server  
Administrator

Password

Services ⓘ

☒ Data  
☒ Index  
☒ Query  
☒ Search

☐ Customize disk storage paths (this node)

## CRUD Example Using CBQ

CBQ, or Couchbase Query, is a query language and command-line tool used to interact with Couchbase data and perform operations such as creating, reading, updating, and deleting data.

Let's assume we will have a Bucket named "Product" having fields "category", "subcategory", "name", and "quantity". First we will have to connect to the Cluster using the following command:

```
C:\Program Files\Couchbase\Server\bin> cbq -e  
http://localhost:8091 -u username -p password
```

Replace the directory with your Couchbase installation directory.

- Create Bucket

```
cbq> CREATE BUCKET Product
```

- Create Index

```
cbq> CREATE PRIMARY INDEX ON `Product`;
```

- Insert Document:

```
cbq> INSERT INTO Product (KEY, VALUE) VALUES ("<document-id>", {  
"category": "<category>", "subcategory": "<subcategory>", "name":  
"<name>", "quantity": <quantity> });
```

- Update existing document:

```
cbq> UPDATE Product SET quantity = <new-quantity> WHERE id =  
"<document-id>"
```

- Read a specific document from the bucket:

```
cbq> SELECT * FROM Product WHERE id = "<document-id>"
```

- Delete a specific document from the bucket using primary key:

```
cbq> DELETE FROM Product WHERE id = "<document-id>"
```

## CRUD Example Using Spring Boot Application

In the `pom.xml` file of our Spring Boot application we need to add the following dependency:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-couchbase</artifactId>
</dependency>
```

The starter project has the `spring-data-couchbase` and `couchbase java-client` in it.

In `application.yml` file add the following configuration properties:

```
spring:
  data:
    couchbase:
      bucket-name: product
      scope-name: _default
      auto-index: true
    couchbase:
      connection-string: couchbase://localhost
      username: ****
      password: *****
```

Create a configuration bean that extends `AbstractCouchbaseConfiguration`:

```
@Configuration
@EnableConfigurationProperties(CouchbaseProperties.class)
@EnableCouchbaseRepositories(basePackages =
    "com.personal.couchbasekafkapoc.repository")
public class CouchbaseConfiguration extends AbstractCouchbaseConfiguration {
    private final CouchbaseProperties couchbaseProperties;
    @Value("${spring.data.couchbase.bucket-name}")
    private String bucketName;
    @Value("${spring.data.couchbase.scope-name}")
    private String scopeName;
    private static final String TYPE_KEY = "type";

    public CouchbaseConfiguration(CouchbaseProperties couchbaseProperties) {
        this.couchbaseProperties = couchbaseProperties;
    }
    @Override
    public String getConnectionString() {
        return couchbaseProperties.getConnectionString();
    }
    @Override
    public String getUsername() {
        return couchbaseProperties.getUsername();
    }
    @Override
```

```

    public String getPassword() {
        return couchbaseProperties.getPassword();
    }
    @Override
    public String getBucketName() {
        return bucketName;
    }
    @Override
    public String typeKey() {
        return TYPE_KEY;
    }
    @Override
    protected String getScopeName() {
        return scopeName;
    }

    @Bean
    public Bucket bucket(Cluster cluster) {
        return cluster.bucket(bucketName);
    }
}

```

Create a model class named Product with appropriate annotations as below:

```

@Document
@TypeAlias(Product.TYPE_NAME)
@Collection(Product.TYPE_NAME)
public class Product implements Serializable {
    public static final String TYPE_NAME = "product";

    @Id
    @GeneratedValue(strategy = UNIQUE, delimiter = ID_DELIMITER)
    private Integer id;
    @Field
    private String category;
    @Field
    private String subcategory;
    @Field
    private String name;
    @Field
    private Integer quantity;

    // Getters & Setters
}

```



Create a Repository that extends *CouchbaseRepository*:

```
@Repository
public interface ProductRepository extends CouchbaseRepository<Product,
Integer> {}
```

The repository provides the implicit CRUD methods.

Create a Service class:

```
@Service
public class ProductService {
    private final ProductRepository productRepository;
    @Autowired
    public ProductService(ProductRepository productRepository) {
        this.productRepository = productRepository;
    }
    public Product createProduct(Product product) {
        return productRepository.save(product);
    }
    public Product updateProduct(Product product) {
        return productRepository.save(product);
    }
    public void deleteProduct(String id) {
        productRepository.deleteById(id);
    }
    public Optional<Product> findProductById(String id) {
        return productRepository.findById(id);
    }
}
```

To test the methods let's create a REST controller:

```
@RestController
@RequestMapping("/products")
public class ProductController {
    private final ProductService productService;
    @Autowired
    public ProductController(ProductService productService) {
        this.productService = productService;
    }
    @PostMapping
    public Product createProduct(@RequestBody Product product) {
        return productService.createProduct(product);
    }
    @PutMapping
    public Product updateProduct(@RequestBody Product product) {
```

```

        return productService.updateProduct(product);
    }
    @DeleteMapping("/{id}")
    public void deleteProduct(@PathVariable String id) {
        productService.deleteProduct(id);
    }
    @GetMapping("/{id}")
    public Optional<Product> findProductById(@PathVariable String id) {
        return productService.findProductById(id);
    }
}

```

We can now run the Spring Boot application and test the CRUD features by invoking the APIs of the controller.

**Reference:**

1. <https://docs.couchbase.com/server/current/learn/architecture-overview.html>
2. <https://www.baeldung.com/spring-data-couchbase>