
Moving From Relational to NoSQL: How to Get Started From Oracle



Couchbase

Why the shift to NoSQL?

As enterprises modernize, teams have to build and maintain applications more rapidly and at greater scale. Applications must be resilient and available whether their clients are web, mobile, or the Internet of Things (IoT). If any of these channels fail, customers go elsewhere. Today, enterprises in every industry from travel, to technology, to retail and services are leveraging NoSQL database technology for more agile development, reduced operational costs, and scalable operations. For many, the use of NoSQL started with a cache, proof of concept (POC), or small application, then expanded to targeted mission-critical applications. NoSQL has now become a foundation for modern web, mobile, and IoT application development.

Some of the largest internet and enterprise companies are using NoSQL technology to deploy their mission-critical applications. Examples include:

- **Marriott** deployed NoSQL to modernize its hotel reservation system that supports \$38 billion in annual bookings.
- **Sky** replaced the limitations of their legacy Oracle RDBMS with the flexibility of NoSQL to improve customer experience during peak traffic.
- **Amadeus** improved query performance by over 1,000% by switching to Couchbase from Oracle

At Couchbase, we've enabled hundreds of enterprises, as well as numerous growth companies and startups, to deploy NoSQL for better agility, performance, and lower costs. The goal of this paper is to help you introduce NoSQL into your infrastructure by highlighting lessons learned from enterprises that successfully adopted NoSQL. We'll explore key considerations and strategies for transitioning to NoSQL, in particular, to a document database (Couchbase Server). We'll provide tips for moving from Oracle and other relational databases. Not every use case requires replacement of an existing RDBMS (relational database management system). There are use cases in which NoSQL is not a replacement for, but a complement to, existing infrastructure, facilitating the use of polyglot persistence.

We'll start with recommendations for identifying and selecting the right application. Next, we'll cover strategies for modeling relational data as documents, how to access them within your application, and how to migrate data from a relational database. Finally, we'll highlight the basics of operating a NoSQL database in comparison to a relational database.





Identifying the right application

Many enterprises have successfully introduced NoSQL by identifying a single application or service to start with. It could be a new one that is being developed or an existing application that's being refactored. Examples include:

- A global service that powers multiple applications
- A logical or physical service within a large application
- A high performance, highly available caching service
- A small, independent application with a narrow scope

Ideal candidates have one or more of the following characteristics or requirements:

- Need to modify the data model continuously or on demand (e.g., to accommodate new preferences, new social media accounts, etc.)
- Ability to read and write JSON documents (semi-structured data) to/from web and mobile clients
- Provide low latency, high throughput access to data
 - e.g., users expect interactions to be instant and waiting negatively impacts the experience (e.g., users abandon shopping carts)
- Support thousands to millions of concurrent users – e.g., the number of users is increasing, sometimes exponentially, as when content goes viral
- Support users in any country or region – they're everywhere
- Support users and be available 24/7
- Store terabytes of data
- Deploy in multiple data centers with an active/active configuration
- Building enterprise applications – with constantly changing schema requirements, complex or unstructured data

Some common examples of good use cases for NoSQL:

- Product catalog service
- Asset tracking service
- Content management service
- Application configuration service
- Customer management service
- File or streaming metadata service

NoSQL has become a foundation for modern web, mobile, and IoT application development.

At Couchbase, we've enabled hundreds of enterprises, as well as numerous growth companies and startups, to deploy NoSQL for better agility, performance, and lower costs.



Modeling and migrating your data

Many enterprises have successfully introduced NoSQL by identifying a single application or service to start with.

Couchbase Server is a **document database** – data is stored in JSON document collections, instead of tables. While relational databases rely on an explicit predefined schema to describe the structure of data, document databases do not – JSON documents are self-describing. As such, every JSON document includes its own flexible schema, and it can be changed on demand by changing the document itself.

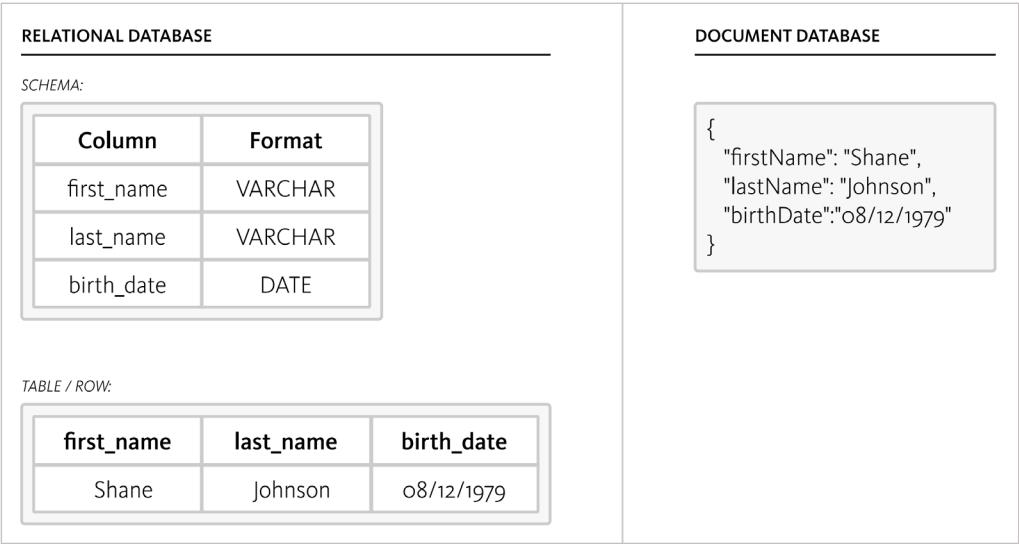


Figure 1: Relational schema and data vs. self-describing JSON documents

It's important to understand that JSON documents are not limited to primitive fields. They can include arrays and objects, and they can be nested, just like applications. For this reason, there is no “impedance mismatch” between application objects and JSON documents. No complex object-relational mapping (ORM) solution is required.

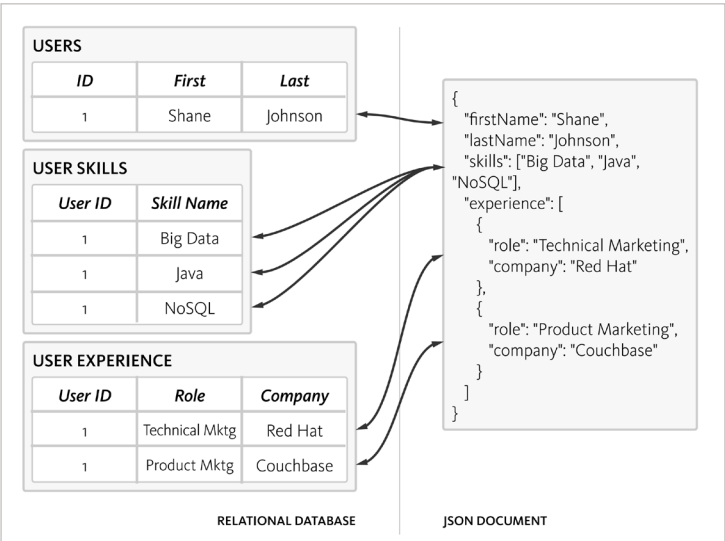


Figure 2: Multiple tables vs. nested data with JSON documents



Just as every row in a table usually has a primary key, every document requires an object ID. Many applications rely on relational databases to automatically generate the primary key (for example, with sequences in Oracle Database). Document databases can use unique keys like UUID/GUID, but applications can also use natural keys where possible.

In a relational database, primary keys are defined per table. It's not uncommon for the primary key of different rows in different tables to have the same value. After all, a row can be identified by its table and primary key. However, document databases do not store documents in tables; in Couchbase Server, they're stored in collections (which are, in turn, stored in scopes, which are, in turn, stored in buckets). You can store any type of document within a collection, but typically you will store similar types of documents within the same collection (i.e., very similar to using a relational table).

Couchbase Server is a document database – data is stored in JSON document collections, not in tables.

Every JSON document includes its own flexible schema, and it can be changed on demand by changing the document itself.

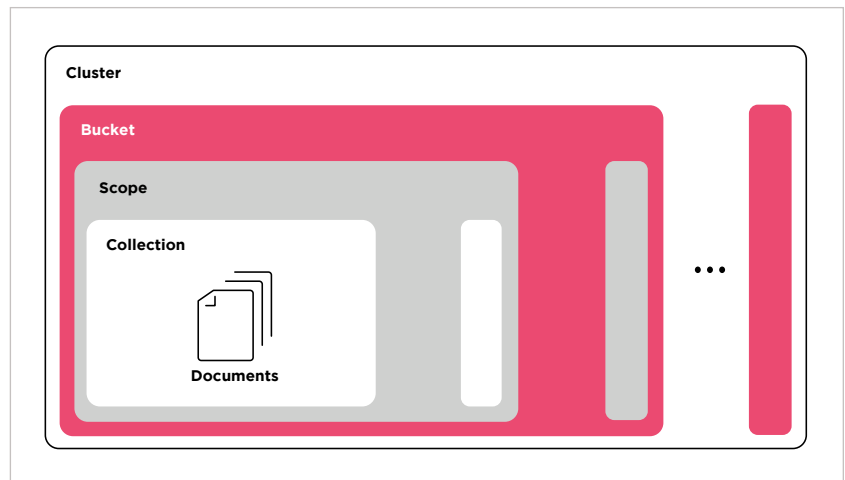


Figure 3: Scopes and Collections Overview

Even with scopes and collections, you can still benefit from natural keys by using them to identify a document by an ID, even if the collection stores different types of documents.

For example, consider a single collection with blogs, authors, and comments stored in separate documents:

- **author::shane**
- **author::shane::blogs**
- **blog::nosql_fueled_hadoop**
- **blog::nosql_fueled_hadoop::comments**

These object IDs not only enable the bucket to store related documents; they're also human readable, deterministic, and semantic. In addition, an application can construct these keys easily to fetch or query using them. Even if you decided to put these four documents into four separate buckets, these keys will still be beneficial.

A document can be modeled after a row (flat), or it can be modeled after related rows in multiple tables (nested). However, documents should be modeled based on how applications interact with the data. While some documents may contain nested data, others may reference it.



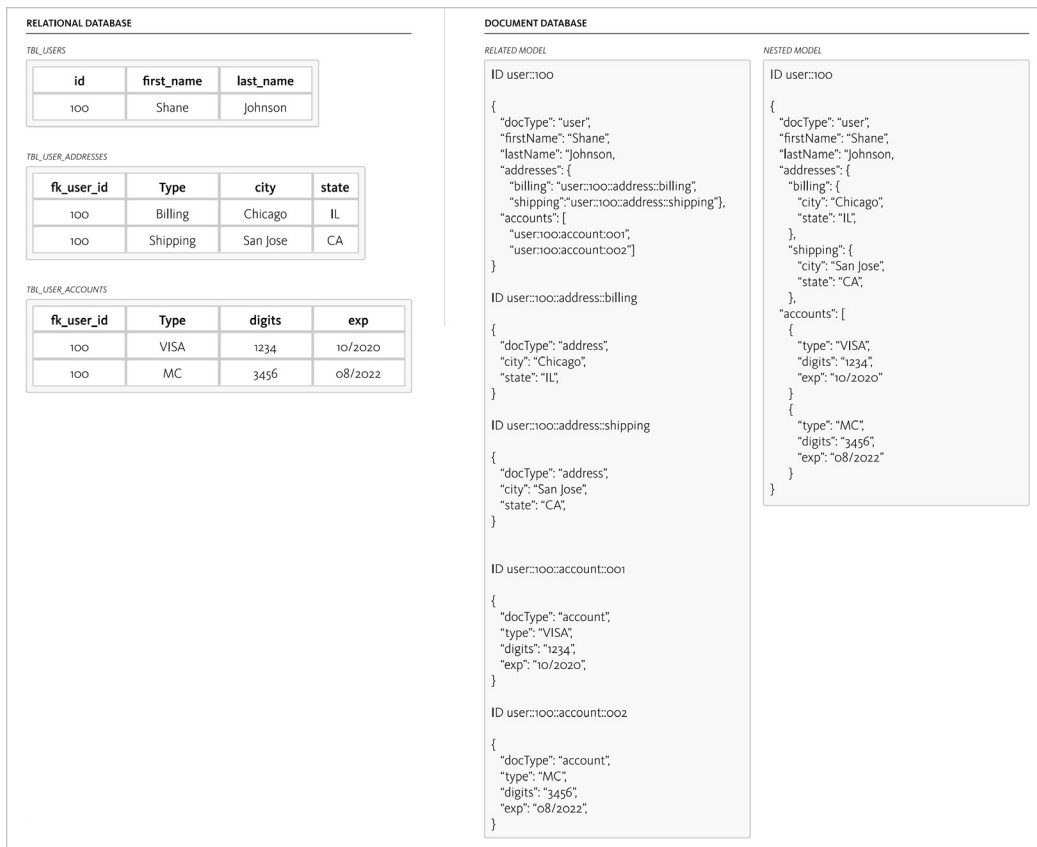


Figure 4: Related vs. nested JSON documents

NOTE: **Couchbase Server** supports atomic counters, and like sequences in Oracle, they can be used to automatically generate object IDs for documents. In addition, counters can be incremented by any amount. Like sequences in Oracle, it's possible for applications to increment the counter by, for example, 50 to avoid making a call to the database on every insert. Instead, the counter can be updated every 50 inserts.

NoSQL databases provide data access via key-value APIs, query APIs, and SDKs. Couchbase supports SQL++ as a query language to its query API.

In relational databases, children always reference their parents via foreign keys. However, in document databases, parents are able to reference their children when appropriate. That's because, while a field in a row can only contain a single value, a field within a document can contain multiple values. In the example of a related model (figure 4), the addresses and accounts fields contain multiple object IDs – one for each address. However, the shipping and billing object IDs are not required – they are deterministic.

A document can contain fields, objects, lists, and arrays. In the examples above, the addresses field contains a list of addresses or references to address. The accounts field contains an array of accounts or references to accounts.

Reference or nest related data?

There are two things to consider when deciding how to model related data:

1. Is it a one-to-one or one-to-many relationship?
2. How often is the data accessed?

MIGRATION TIP: Create object IDs for documents that include the row's primary key. For example, if the primary key of a row in the products table is 123, the document ID is product::123. However, if you are putting documents in a product collection, you may want to omit "product::" to save memory and disk space.

If it's a one-to-one or one-to-many relationship (a child has one parent), it may be better to store the related data as nested objects. This approach results in a simple data model and reduces or eliminates the need to query multiple documents. However, if it's a many-to-one or many-to-many relationship (a child has multiple parents), it may be better to store the related data as separate documents, which reduces or eliminates the need to maintain duplicate data.

If a majority of the reads are limited to parent data (e.g., first and last name), it may be better to model the children (e.g., addresses and accounts) as separate documents. This results in better performance because the data can be read with a single key-value operation instead of a query, and reduces bandwidth because the amount of data being transferred is smaller. However, if a majority of the reads include both parent and child data, it may be better to model the children as nested objects. This approach results in great performance because the data can be read with a single key-value operation instead of a query.

If a majority of the writes are to the parent or child, but not both, it may be better to model the children as separate documents. For example, if user profiles are created with a wizard – first add info, then add addresses, finally add accounts – or if a user can update an address or account without updating their info. However, if a majority of writes are to parent and child (both) – for example, there's a single form to create or update a user – it may be better to model the children as nested objects.

When to nest? Considerations:

- One-to-one or one-to-many? Nest
- Many-to-one or many-to-many? Don't nest.
- Most reads are for parent data? Don't nest.
- Most reads are for parent and child together? Nest.
- Most writes are for parent or child? Don't nest.
- Most writes are for parent and child together? Nest.

Finally, it may be better to model children as separate documents to reduce document size and write contention. For example, the number of reviews on a product may grow indefinitely. If they were embedded, the size of the product document could become excessive, resulting in slower reads. Consider a blog and comments. When a blog is first published, there may be a lot of readers posting comments. If the comments are embedded, many concurrent users will be trying to update the same blog document at the same time, resulting in slower writes. A good compromise may be to store comments as separate threads – a document for every top-level comment that embeds all replies.

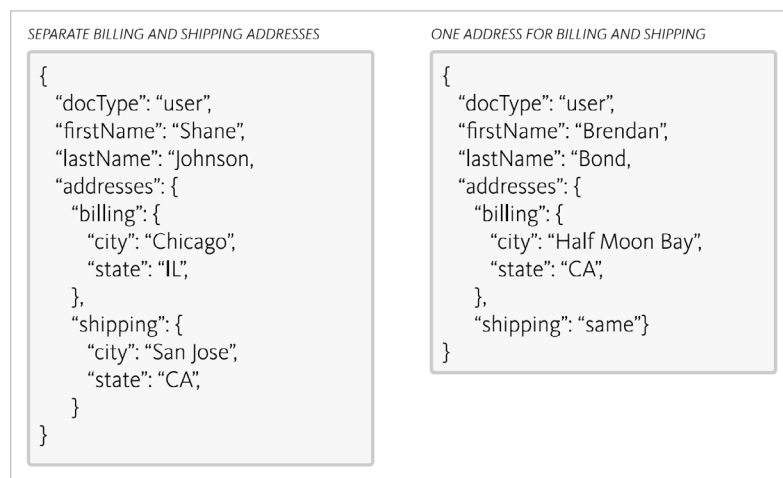


Figure 5: Different documents of the same type can have different schema



Performing a migration?

NOTE: Same object, different structure?

As illustrated in figure 5, it's possible for the same field or object to have a different structure in different documents.

When making your modeling decisions, remember that Couchbase supports both SQL JOINS and ACID transactions. If you need to model data separately, you will still be able to operate on it together.

The easiest and fastest way to get started is to export your relational data to CSV files, and import them into Couchbase Server. This may not represent the final data model, but it will enable you to start interacting with Couchbase Server right away. Couchbase Server includes a command-line utility, `cbimport`, for importing data in CSV files.

```
$ cbimport csv -c couchbase://127.0.0.1 -u Administrator -p
password -b default --scope collection-exp myscope.products -d
file:///products.csv -g %id% -t 4
```

Understanding your access patterns

NoSQL databases provide data access via key-value APIs, SQL++ query APIs, or SDKs. The key-value API provides the best performance – since it will often be direct from an in-memory cache. The query API or language provides the most power and flexibility – enabling applications to sort, filter, transform, group, and combine documents. Queries to Couchbase are done using SQL, just like in a relational database (with extensions for JSON, hence “SQL++”).

Key value

The key-value API can provide a great deal of data access without the need to perform queries. In the example below, once you have the object ID of the user profile document, you can figure out what the object IDs of the address and account documents are.

OPTION ONE: RELATED DATA IS REFERENCED

Get my profile

Get my billing address

Update my shipping address

Add a credit card

```
1 bucket.get("user::100");
2
3 bucket.get("user::100::address::billing");
4
5 bucket.replace("user::100::address::shipping", address);
6
7 bucket.insert("user::100::account::002, account);
8
```

OPTION TWO: RELATED DATA IS NESTED

Get my profile

Get my billing address

Update my shipping address

Add a credit card

```
1 bucket.get("user::100");
2
3 profile.getObject(addresses.billing);
4
5 // replace the old shipping address
6 profile.put("addresses.shipping", address);
7
8 // update the database
9 bucket.replace("user::100", profile);
10
11 // get the accounts array
12 profile.getArray("accounts");
13
14 // add the new account to the array
15 accounts.add(account);
16
17 // update the database
18 bucket.replace("user::100", profile);
19
```



Query

The query API or language, combined with proper indexing, can provide a great deal of power and flexibility without sacrificing performance. Couchbase Server provides a SQL++ implementation called NIQL, which extends SQL to JSON documents. NIQL also has support for ANSI joins making it easier for developers to apply their SQL knowledge to develop applications within Couchbase.

OPTION ONE: RELATED DATA IS REFERENCED

Get all Platinum users

Get all users with a Visa or Mastercard account

Get all users with a billing address in California

```
1  SELECT      firstName, lastName
2  FROM        users
3  WHERE       status = "Platinum";
4
5  SELECT      firstName, lastName
6  FROM        users u INNER JOIN accounts a
7  ON KEYS     u.accounts
8  WHERE       a.type = "Visa" OR
9              a.type = "Mastercard";
10
11 SELECT      firstName, lastName
12 FROM        users u INNER JOIN addresses a
13 ON KEYS     u.addresses.shipping
14 WHERE       a.state = "CA";
15
```

OPTION TWO: RELATED DATA IS NESTED

Get all Platinum users

Get all users with a Visa or Mastercard account

Get all users with a billing address in California

```
1  SELECT      firstName, lastName
2  FROM        users
3  WHERE       status = "Platinum";
4
5  SELECT      firstName, lastName
6  FROM        users
7  WHERE ANY   account IN users.accounts
8  SATISFIES  account.type = "Visa" OR
9              account.type = "Mastercard" END;
10
11 SELECT      firstName, lastName
12 FROM        users
13 WHERE       users.addresses.billing.state = "CA";
14
```

While one of the benefits of storing related data as separate documents is the ability to read a subset of the data (e.g., shipping address), the same thing can be accomplished with a query API or language when related data is nested. For example, to read the billing address from a user profile document that stores all related data as nested objects.

```
1  SELECT      addresses.billing
2  FROM        users
3  USE KEYS    ["user::100"];
4
```

In addition, while one of the benefits of storing related data as nested objects is the ability to access all data with a single read, the same thing can be accomplished with a query API or language when related data is stored as separate documents. For example, to read the user profile and accounts and addresses when they are stored as separate documents.



SQL++ abstracts the data model from the application model. Regardless of how data is modeled in the database, applications can query it any way they need to by joining documents, nesting and unnesting them, and more. It provides developers with the flexibility they need to model data one way and query it in many.

```
1 SELECT *
2 FROM users
3 NEST accounts ON KEYS users.accounts
4 NEST shippingAddress
5 ON KEYS users.addresses.shipping
6 NEST billingAddress
7 ON KEYS users.addresses.billing
8 USE KEYS ["user::100"];
9
```

The query language can be used to perform CRUD operations as an alternative to the key-value API. This enables applications built on top of a relational database to migrate all data access by replacing SQL statements with SQL++ statements. CRUD operations with SQL++ have the ability to perform partial updates:

Create user

Get user

Update user

Delete user

```
1 INSERT INTO users (KEY, VALUE)
2 VALUES (
3     "user::100",
4     {
5         "firstName": "Shane",
6         "lastName": "Johnson"
7     }
8 );
9
10 SELECT *
11 FROM users
12 USE KEYS "user::100";
13
14 UPDATE users
15 USE KEYS "user::100"
16 SET status = "Platinum";
17
18 DELETE FROM users
19 USE KEYS "user::100";
20
```

Couchbase Server's implementation of the SQL++ standard is called NIQL, which extends SQL to JSON documents.

SQL++ abstracts the data model from the application model. Regardless of how data is modeled in the database, applications can query it any way they need to by joining documents, nesting and unnesting them, and more. It provides developers with the flexibility they need to model data one way and query it in many.

While it's possible to migrate or develop a new application by modeling rows as documents and SQL queries as SQL++ queries, use the key-value approach as much as you can, and rely on SQL++ queries when you can't. When the ID is known, or can be determined, key-value operations provide the best possible performance. SQL++ queries provide the most flexibility by enabling developers to query data regardless of how it's modeled.



Indexing your data

Query performance can be improved by indexing data. NoSQL databases support indexes to varying degrees

– Couchbase Server includes comprehensive indexing support. Below are some indexing examples.

A **simple index** on the user status:

INDEX

```
1 CREATE INDEX idx_user_status
2 ON      users(status);
3
```

QUERY

```
1 SELECT count(status) as count
2 FROM   users
3 WHERE  status = "Platinum";
```

A **composite index** on user status and shipping state:

INDEX

```
1 CREATE INDEX idx_user_status_state
2 ON      users(status, addresses.shipping.state);
3
```

QUERY

```
1 SELECT lastName, firstName
2 FROM   users
3 WHERE  status = "Platinum" AND
4        addresses.shipping.state = "CA";
```

Functional index on shipping state:

INDEX

```
1 CREATE INDEX idx_state
2 ON      users(UPPER(addresses.shipping.state));
3
```

QUERY

```
1 SELECT lastName, firstName
2 FROM   users
3 WHERE  UPPER(addresses.shipping.state) = "CA";
```



Partial index on user billing state of users with a Visa credit card:

INDEX	
1	CREATE INDEX idx_user_account_visa
2	ON users(UPPER(addresses.billing.state))
3	WHERE accounts.type="VISA";
QUERY	
1	SELECT lastName, firstName
2	FROM users
3	WHERE UPPER(addresses.billing.state) = "CA"
4	AND accounts.type="VISA";

Array index on phones:

INDEX	
1	CREATE INDEX idx_phone_number
2	ON users(DISTINCT phones);
3	
QUERY	
1	SELECT lastName, FirstName
2	FROM users
3	WHERE ANY v IN phones SATISFIES v IN ["408.123.1212", "510.121.2122"] END

Couchbase Server supports index intersection. A query can scan multiple indexes in parallel. As a result, it may not be necessary to create multiple indexes that include the same field, thereby reducing both disk and memory usage. You can also index a large number of documents and horizontally scale out an index as needed. The system will transparently partition the index **across a number of index nodes** using hash partitioning and will increase the performance and data capacity of the cluster. For more index examples please see the documentation [here](#).





Connecting to the database

Applications access data in NoSQL databases via clients. Couchbase Server clients are topology-aware (e.g., smart clients) and are available in many languages/platforms: Java, .NET, Node.js, PHP, Python, C, and more.

RELATIONAL	COUCHBASE SERVER
<pre>1 SQLServerDataSource ds = new SQLServerDataSource(); 2 3 ds.setServerName("localhost"); 4 ds.setPortNumber(1433); 5 6 Connection conn = ds.getConnection(); 7 8 <i>// access data via a connection</i> 9 </pre>	<pre>1 List<String> nodes = Arrays.asList("127.0.0.1"); 2 3 CouchbaseCluster cluster = 4 CouchbaseCluster.create(nodes); 5 6 Bucket bucket = cluster.getBucket("users"); 7 8 <i>// access data via a bucket</i> 9 </pre>

Figure 6: Creating a connection to a relational database vs. Couchbase Server

NoSQL databases support indexes to varying degrees – Couchbase Server includes comprehensive indexing support.

A bucket is a higher-level abstraction than a connection, and a cluster can contain multiple buckets. In the example above, the application can access data in the “users” bucket. However, while key-value operations are limited to scopes and collections within the users bucket, SQL++ queries can query between multiple buckets when necessary.

Couchbase Server is a distributed database, but applications do not have to pass in the IP address of every node. However, they should provide more than one IP address so that if the first node is unavailable or unreachable, they can try to connect to the next node.

After the client connects to a node, it will retrieve the IP address of the remaining nodes.

Couchbase Server clients also maintain a cluster map, which enables them to communicate directly with nodes. In addition, the cluster map enables operations teams to scale out the database without impacting the application. Regardless of the number of nodes, the application sees a single database. There are no application changes required to scale from a single node to dozens – the clients are automatically updated.

In addition, with Couchbase Server, applications no longer have to rely on object-relational mapping frameworks for data access, because there is no impedance mismatch between the data model and the object model. In fact, domain objects are optional. Applications can interact with the data via document objects or by serializing domain objects to and from JSON.



DOMAIN OBJECTS

```
1 | JSONDocument doc1 = bucket.get("user::100");
2 |
3 | String json1 = doc.content().toString();
4 |
5 | User user = fromJson(json, User.class);
6 |
7 | // update user status
8 | user.setStatus("Gold");
9 |
10 | String json2 = toJson(user);
11 |
12 | JSONObject obj = JSONObject.from(json2);
13 |
14 | JSONDocument doc2 =
15 |     JSONDocument.create(doc1, obj);
16 |
17 | // update the database
18 | bucket.replace("user::100", doc2);
```

DOCUMENT OBJECTS

```
1 | JSONDocument doc = bucket.get("user::100");
2 |
3 | JSONObject user = doc.content();
4 |
5 | // update user status
6 | user.put("status", "Gold");
7 |
8 | // update the database
9 | bucket.replace("user::100", doc);
10 |
```

Figure 7: Working with domain objects vs document objects

Applications access data in NoSQL databases via clients.

Couchbase Server's topology-aware clients (e.g., smart clients) are available in many languages/platforms: Java, Node.js, .NET, PHP, Python, C, and more.

NOTE: In addition to the clients, there are supported, certified JDBC/ODBC database drivers available for Couchbase Server.

It's easy to serialize domain objects to and from JSON, and it may be helpful to do so for applications with a complex domain model or business logic. However, for new applications or services, working with document objects will require less code and provide more flexibility – developers can change the data model without having to change the application model. For example, you can add a new field to a form without changing application code.

Transactions

Similar to a relational database, Couchbase offers ACID transaction support. Transactions must be initiated by the Couchbase SDK, and can include key-value operations and/or SQL operations.

Transactions are not required when changing data in a single document structure as changes within a document are atomic. When architecting new data structures this should be taken into account as atomic changes to a single document are lighter weight.




```

transactions.run((ctx) -> {
    // Inserting a doc:
    String docId = "aDocument";
    ctx.insert(collection, docId, JsonObject.create());

    // Getting documents:
    // Use ctx.getOptional if the document may or may not exist
    Optional<TransactionGetResult> docOpt = ctx.getOptional(collection, docId);

    // Use ctx.get if the document should exist, and the
    // transaction will fail if not
    TransactionGetResult doc = ctx.get(collection, docId);

    // Replacing a doc:
    TransactionGetResult anotherDoc = ctx.get(collection, "anotherDoc");
    // TransactionGetResult is immutable, so get its content
    // as a mutable JsonObject
    JsonObject content = anotherDoc.contentAs(JsonObject.class);
    content.put("transactions", "are awesome");
    ctx.replace(anotherDoc, content);

    // Removing a doc:
    TransactionGetResult yetAnotherDoc = ctx.get(collection,
                                                    "yetAnotherDoc");

    ctx.remove(yetAnotherDoc);
    ctx.commit();
});
} catch (TransactionFailed e) {
    System.err.println("Transaction " + e.result().transactionId() + " failed");

    for (LogDefer err : e.result().log().logs()) {
        System.err.println(err.toString());
    }
}
}

```

Figure A: Transactions using the Java synchronous API

Couchbase also provides an asynchronous API. It is possible to use transactions asynchronously.

```

transactions.run((ctx) -> {
    TransactionGetResult anotherDoc = ctx.get(collection, "anotherDoc");
    JsonObject content = anotherDoc.contentAs(JsonObject.class);
    content.put("transactions", "are awesome");
    ctx.replace(anotherDoc, content);
});

```

Figure B: Transactions using the Java synchronous API



Installing and scaling your database

One of the key advantages driving the adoption of NoSQL databases with a distributed architecture is their ability to scale faster, easier, and at a significantly lower cost than relational databases. While most relational databases are capable of clustering (e.g., Microsoft SQL Server), they are still limited to scaling up – failover clustering relies on shared storage while always-on availability groups are limited to replication. As a result, more data requires a bigger disk, and more users require a bigger server. The shared storage not only becomes a bottleneck, it becomes a single point of failure. In contrast, most NoSQL databases are distributed to scale out – more data requires more disks, not a bigger one, and more users require more servers, not a bigger one.

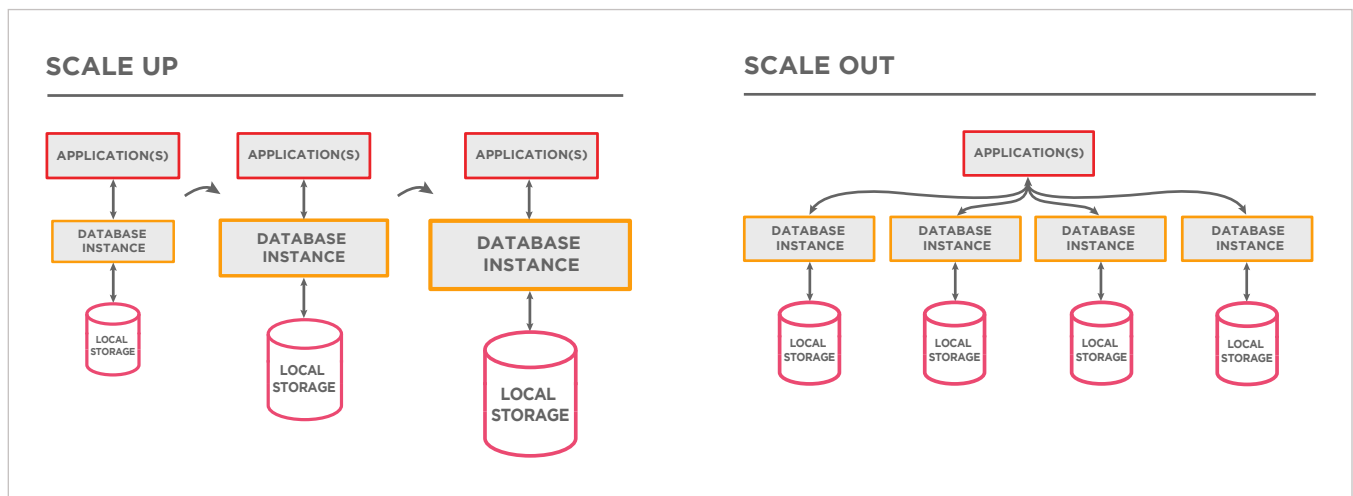


Figure 8: Scaling up vs. scaling out

Couchbase Server's topology-aware clients and consistent hashing distribute data evenly within a cluster. In addition, data can be replicated to one or more nodes to provide high availability. All of these functions are transparent to the developer and handled automatically by Couchbase. You do not need to create/maintain shards or make separate writes for replication.



Installing Couchbase Server

One of the key advantages driving adoption of NoSQL databases with a distributed architecture is their ability to scale faster, easier, and at significantly lower cost than relational databases.

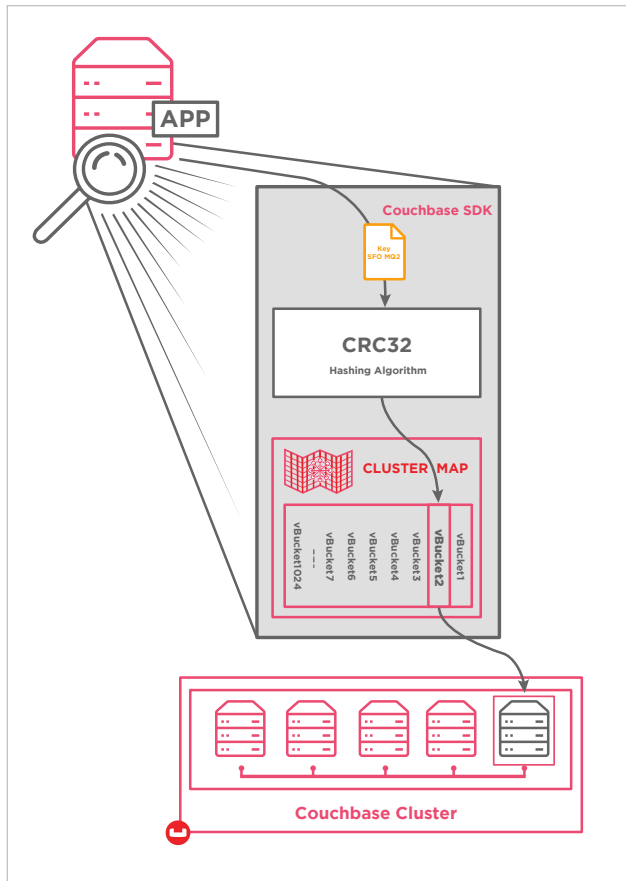


Figure 9: Topology-aware clients in Couchbase Server

Installing Couchbase Server requires little more than downloading the install binary, running it, and configuring the database via the web-based administrative console or CLI. It can also be installed via Docker.

Perhaps the easiest way to get started is with Couchbase's Database-as-a-Service (DBaaS), Couchbase Capella™. You can start a free trial [here](#).

Couchbase delivered as a service

With Capella, database management (setup, ongoing operations, and maintenance) is automated and streamlined so customers can focus on other areas like application development and improving time to market. Key benefits include:

Fully managed

- Automated setup, backups, upgrades, and ongoing management to deliver an always-on service, reducing your operational efforts.

Automated scaling in/out and up/down

- Easily add, remove, or change nodes to meet your current needs. Couchbase rebalances your data. No application changes needed.

Single pane for multi-cluster, multi-region

- The Capella control plane manages data across clusters and across clouds, allowing you to be cloud-provider agnostic. It also provides tools for SQL++ access, document viewing, index creation, and full-text search within the UI.

High availability

- Capella guarantees the global reliability of your data throughout regions and availability zones via native replication across geo-aware clusters, all day, every day.

Self-monitoring, self-healing

- Capella proactively monitors clusters 24/7 to locate, assess, and resolve issues automatically.

Security peace of mind

- In addition to SOC II compliance, Capella delivers end-to-end encryption from the SDK to the disk and offers granular role-based access control.





Monitoring and managing your deployment

Couchbase Server and Couchbase Capella include an integrated, comprehensive administration console as well as REST and CLI APIs.

While many relational and NoSQL databases require separate administration tools, Couchbase Server and Couchbase Capella include an integrated, comprehensive administration console as well as REST and CLI APIs.

The administration console and the REST/CLI APIs enable administrators to manage and monitor clusters, both small and large, with minimal effort. Functionality enabled through the Couchbase admin console includes:

Management, monitoring, and configuration

- Cluster/node/bucket
- Cross data center replication (XDCR)
- Database performance, network utilization, resource utilization

Tasks

- Add and remove nodes
- Failover nodes
- Rebalance cluster

Configuration

- Authentication and authorization
- Auditing

Monitor

- View and collect log information

In addition to the administration console and APIs, Couchbase Server includes a number of command line tools to perform additional tasks such as, among others:

- `cbbackupmgr` – full, cumulative, and incremental backup and restore
- `cbcollect_info` and `cbdstats` – gather node and cluster diagnostics (280+ metrics)
- `cbq` – run SQL++ queries from the command line
- `cbimport/cbexport` – transfer data to and from JSON or CSV files



Putting it all together: How to conduct a successful POC

Now that you're familiar with the key considerations and strategies for transitioning from a relational database to a NoSQL database, you're ready to start a proof of concept (POC).

Now that you're familiar with the key considerations and strategies for transitioning from a relational database to a NoSQL database – how to select an application, how to model and access the data, and how to deploy the database – you're ready to start a proof of concept.

Couchbase solutions engineers have helped, and continue to help, many enterprises successfully introduce NoSQL, from planning all the way to post-production. We encourage everyone to start with a proof of concept.

There are five steps to a successful proof of concept:

1. Select a use case and application

It's important to remember that the key to successfully introducing a NoSQL database is to first identify an appropriate use case and select an application. Which application can realize one or more of the benefits of a NoSQL database: better performance and scalability, higher availability, greater agility, and/or improved operational management.

2. Define the success criteria

It may be difficult to move beyond a proof of concept without defining how to measure its success. Success criteria vary for different applications. For some, it may be performance (e.g., 5ms latency in the 95th percentile). For others, it may be management (e.g., easier to scale and add nodes). It may be faster development cycles. Whatever it may be, make sure to specify it upfront.

3. Understand the data

Before defining the data model, simply understand the data and the business domain. At first, the focus should not be on how to define or migrate the data model. Rather, it should be on understanding the data, independent of how it's stored in the database.

4. Identify the access patterns

Next, identify how the data is used and then begin to model it within a NoSQL database. This will depend very much on how the application reads, writes, and finds data. The data model can be optimized for different access patterns. In addition, you have to choose the right data access method – key-value operations, SQL++ queries, full-text search, analytics, etc. – for the right data access pattern – basic read/write operations, queries, or aggregation and reporting.

5. Review the architecture

After completing the proof of concept and measuring the results against its predefined success criteria, it's time to begin preparing for production deployment. This is the time to review the architecture and create a blueprint for production. Based on the PoC development experience, you can identify what worked well and what could work better – use this knowledge to help define the final application architecture.



NoSQL success offers rich rewards

NoSQL was expressly designed for the requirements of modern web, mobile, and IoT applications. For enterprises that make the shift to NoSQL, the rewards are significant: greater agility, faster time to market, easier scalability, better performance and availability, and lower costs. Developers find that working with a JSON data model is far more natural than a rigidly defined relational schema, especially since Couchbase retains some familiar features of the relational world (SQL and ACID transactions). Operations engineers love the ease of elastically scaling the database without all the headaches of manual sharding and skyrocketing costs.

If you're ready to take the next steps and are looking for more specific advice, we invite you to talk with one of our solutions engineers. At a minimum, you'll probably get some helpful insights and best practices for your particular use case.

For enterprises that make the shift to NoSQL, the rewards are significant: greater agility, faster time to market, easier scalability, better performance and availability, and lower costs.





At Couchbase, we believe data is at the heart of the enterprise. We empower developers and architects to build, deploy, and run their mission-critical applications. Couchbase delivers a high-performance, flexible and scalable modern database that runs across the data center and any cloud. Many of the world's largest enterprises rely on Couchbase to power the core applications their businesses depend on.

For more information, visit www.couchbase.com.

© 2022 Couchbase. All rights reserved.