Moving from Relational to NoSQL: How to Get Started
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.

Why the shift to NoSQL?

As enterprises shift to the Digital Economy – an economy powered by cloud, mobile, social media, and big data technologies – development and operations teams have to build and maintain web, mobile, and Internet of Things (IoT) applications faster and faster, and at greater scale. Increasingly, these applications are how companies interact with customers, employees, partners, and products.

Today, enterprises in every industry from Internet to Retail to 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.

Do a quick Google search for “NoSQL” plus the name of any major enterprise known for IT excellence and you’ll see they’re deploying NoSQL for many mission-critical applications. Examples abound:

- **Gannett**, publisher of USA Today and 90+ media properties, replaced relational database technology with NoSQL to power its digital publishing platform.

- **Marriott** deployed NoSQL to modernize its hotel reservation system that supports $38 billion in annual bookings.

- **Ryanair** moved to NoSQL to power its mobile app that serves over 3 million passengers.

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), with tips for moving from Oracle and other relational databases. Note, 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.
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 high performance, highly available caching service
- A small, independent application with a narrow scope
- A logical or physical service within a large application
- A global service that powers multiple applications

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

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

- Fast iteration – the data model needs to be modified continuously or on demand (e.g., to accommodate new preferences, new social media accounts, etc.)
- 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
- Users are not limited to a specific country or region – they’re everywhere
- Users are not limited to specific hours – the service has to be available 24x7
- Store terabytes of data
- Deploy in multiple datacenters with an active/active configuration

**Some common examples of good fits for NoSQL:**

- Product catalog service
- Asset tracking service
- Content management service
- Application configuration service
- Customer management service
- File or streaming metadata service
Modeling and migrating your data

Couchbase Server is a **document database** — data is stored in JSON documents, not in 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](image1.png)

It’s important to understand that JSON documents are not limited to 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.

![Figure 2: Multiple tables vs. nested data with JSON documents](image2.png)

Just as every row in a table requires 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). However, with document databases, applications should leverage 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 is identified by its table and primary key. However, document databases do not store documents in tables; in Couchbase Server, they’re stored in “buckets.” A bucket can store different types of documents (i.e., database), or it can store the same type of documents (i.e., table).

**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.

The benefit of using natural keys with a document database is that a document can be identified by an object ID, even if the bucket stores different types of documents.

For example, consider a single bucket 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.

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.
In the absence of tables, applications can benefit from including a field in documents that identifies the type of the document. In the example of figure 4, there are user, address, and account types. The type field can then be indexed to improve query performance when multiple types of documents are stored within the same bucket.

In relational databases, children reference their parents via foreign keys. However, in document databases, parents reference their children. 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.

---

**Figure 4: Related vs. nested JSON documents**

In the absence of tables, applications can benefit from including a field in documents that identifies the type of the document. In the example of figure 4, there are user, address, and account types. The type field can then be indexed to improve query performance when multiple types of documents are stored within the same bucket.

In relational databases, children reference their parents via foreign keys. However, in document databases, parents reference their children. 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?
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.

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.

Performing a migration?

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, cbtransfer, for importing data in CSV files. By default, it assumes the first row is the field names and one of them is “id”.

```
$ cbtransfer /<path to file>/<filename>.csv http://<cluster ip>:8091 -B <bucket name> -u Administrator -p password
```

Figure 5: Different documents of the same type can have different schema
Understanding your access patterns

NoSQL databases provide data access via key-value APIs, query APIs or languages, and MapReduce. The key-value API provides the best performance. The query API or language provides the most power and flexibility – enabling applications to sort, filter, transform, group, and combine documents. MapReduce is ideal for performing aggregation over a large number of documents, if not all of them.

Key Value

The key-value API, when combined with proper key naming conventions, 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

```java
/* get my profile */
bucket.get("user::100");

/* get my billing address */
bucket.get("user::100::address::billing");

/* update my shipping address */
bucket.replace("user::100::address::shipping", address);

/* add a credit card */
bucket.insert("user::100::account::002", account);
```

**Option two:** related data is nested

```java
/* get my profile */
bucket.get("user::100");

/* get my billing address */
profile.getObject(addresses.billing);

/* update my shipping address */
// replace the old shipping address
profile.put("addresses.shipping", address);

/* update my shipping address */
// update the database
bucket.replace("user::100", profile);

/* add a credit card */
// get the accounts array
profile.getArray("accounts");

/* add a credit card */
// add the new account to the array
accounts.add(account);

/* add a credit card */
// update the database
bucket.replace("user::100", profile);
```
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 query language called N1QL, which extends SQL to JSON documents. N1QL also has support for ANSI joins making it easier for developers to leverage their SQL knowledge to develop applications with the Couchbase Data Platform.

Option one: related data is referenced

```sql
/* get all Platinum users */
SELECT  firstName, lastName
FROM  users
WHERE  status = "Platinum";

/* get all users with a Visa or Mastercard account */
SELECT  firstName, lastName
FROM  users u INNER JOIN accounts a
ON KEYS u.accounts
WHERE  a.type = "Visa" OR
       a.type = "Mastercard";

/* get all users with a billing address in California */
SELECT  firstName, lastName
FROM  users u INNER JOIN addresses a
ON KEYS u.addresses.shipping
WHERE  a.state = "CA";
```

Option two: related data is nested

```sql
/* get all Platinum users */
SELECT  firstName, lastName
FROM  users
WHERE  status = "Platinum";

/* get all users with a Visa or Mastercard account */
SELECT  firstName, lastName
FROM  users
WHERE ANY account IN users.accounts
      SATISFIES account.type = "Visa" OR
            account.type = "Mastercard" END;

/* get all users with a billing address in California */
SELECT  firstName, lastName
FROM  users
WHERE  users.addresses.billing.state = "CA";
```
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"];  
```

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.

```
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"];  
```

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 N1QL statements. One of the advantages of performing CRUD operations with a query language is the ability to perform partial updates:

```
/* create user */
INSERT INTO users(KEY,VALUE) VALUES  
  (  
    "user::100",  
    "firstName": "Shane",  
    "lastName": "Johnson",  
  )
;

/* get user */
SELECT * FROM   users USE KEYS  "user::100";

/* update user */
UPDATE   users USE KEYS  "user::100" SET   status = "Platinum";

/* delete user */
DELETE FROM   users USE KEYS  "user::100";
```
N1QL 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.

**MapReduce**

Many NoSQL databases include a MapReduce implementation. Couchbase Server implements incremental MapReduce with views. A view is similar to a materialized view in a relational database – in particular, when a materialized view is created to aggregate data for queries.

Consider the following example:

<table>
<thead>
<tr>
<th>TABLE</th>
<th>MATERIALIZED VIEW</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRB_USERS</td>
<td></td>
</tr>
<tr>
<td>first_name</td>
<td>state</td>
</tr>
<tr>
<td>Lija</td>
<td>CA</td>
</tr>
<tr>
<td>Shane</td>
<td>CA</td>
</tr>
<tr>
<td>Adam</td>
<td>FL</td>
</tr>
<tr>
<td>Dustin</td>
<td>FL</td>
</tr>
<tr>
<td>Keith</td>
<td>FL</td>
</tr>
<tr>
<td>Lisa</td>
<td>NC</td>
</tr>
<tr>
<td>Rob</td>
<td>NC</td>
</tr>
</tbody>
</table>

When applications query the materialized view, they are querying pre-computed data. The same is true when applications query views in Couchbase Server. While relational databases generate materialized views with a SQL query, Couchbase Server generates views with MapReduce.

<table>
<thead>
<tr>
<th>MAP / REDUCE</th>
<th>VIEW</th>
</tr>
</thead>
<tbody>
<tr>
<td>//Map</td>
<td>STATE_USERS</td>
</tr>
<tr>
<td>function(doc, meta) {</td>
<td>{</td>
</tr>
<tr>
<td>emit(doc.state, 1);</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td>//Reduce</td>
<td></td>
</tr>
<tr>
<td>function(key, values, rereduce) {</td>
<td></td>
</tr>
<tr>
<td>if (rereduce) {</td>
<td></td>
</tr>
<tr>
<td>var result = 0;</td>
<td></td>
</tr>
<tr>
<td>for (var i = 0; i &lt; values.length; i++) {</td>
<td></td>
</tr>
<tr>
<td>result += values[i];</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>return result;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>else {</td>
<td></td>
</tr>
<tr>
<td>return values.length;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

Couchbase Server views are sorted based on the key – in the example above, the state – and support multiple queries.
/* get the number of users per state */
bucket.query(
  ViewQuery.from("designdoc", "state_users"));

/* get the number of users in California */
bucket.query(
  ViewQuery.from("designdoc", "state_users")
  .key("CA"));

/* get the number of users in every state from AL to IL */
bucket.query(
  ViewQuery.from("designdoc", "state_users")
  .startKey("AL").endKey("IL"));

Couchbase Server views can be used for roll up aggregation as well. For example, a
view could be created to maintain the number of new users per year, month, and day.
Applications can then submit view queries that group it by year, year + month, or year + month + day. In addition, Couchbase Server views can be used to create secondary
indexes based on complex logic.

The difference is that Couchbase Server views are updated automatically and in
increments. Imagine if materialized views were automatically updated every few seconds
by performing the query on recently created or updated rows, and merging the results.
That’s how Couchbase Server updates views.

While it’s possible to migrate or develop a new application by modeling rows as
documents and SQL queries as N1QL queries, the best approach is to leverage the
key-value API, N1QL queries, and MapReduce views. When the ID is known, or can
be determined, key-value operations provide the best possible performance. When
performing aggregation over a large, if not entire, dataset, MapReduce provides the
best possible performance. Finally, N1QL 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:

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

QUERY
1 SELECT count(status) as count
2 FROM users
3 WHERE status = "Platinum";
```
NoSQL databases support indexes to varying degrees - Couchbase Server includes comprehensive indexing support.

A composite index on user status and shipping state:

```sql
CREATE INDEX idx_user_status_state
ON users(status, addresses.shipping.state);
```

```sql
SELECT lastName, firstName
FROM users
WHERE status = "Platinum" AND addresses.shipping.state = "CA";
```

Functional index on shipping state:

```sql
CREATE INDEX idx_state
ON users(UPPER(addresses.shipping.state));
```

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

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

```sql
CREATE INDEX idx_user_account_visa
ON users(UPPER(addresses.billing.state))
WHERE accounts.type = "VISA";
```

```sql
SELECT lastName, firstName
FROM users
WHERE UPPER(addresses.billing.state) = "CA"
AND accounts.type = "VISA";
```

Array index on phones:

```sql
CREATE INDEX idx_phone_number
ON users(DISTINCT phones);
```

```sql
SELECT lastName, FirstName
FROM users
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.
Connecting to the database

Applications access data in NoSQL databases via clients. Couchbase Server leverages topology-aware clients (e.g., smart clients) available in many languages: Java, Node.js, PHP, Python, C, and more. These clients are configured in much the same way JDBC/ODBC drivers are configured.

<table>
<thead>
<tr>
<th>RELATIONAL</th>
<th>COUCHBASE SERVER</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLServerDataSource ds = new SQLServerDataSource();</td>
<td>List&lt;String&gt; nodes = Arrays.asList(&quot;127.0.0.1&quot;);</td>
</tr>
<tr>
<td>ds.setServerName(&quot;localhost&quot;);</td>
<td>CouchbaseCluster cluster = CouchbaseCluster.create(nodes);</td>
</tr>
<tr>
<td>ds.setPortNumber(1433);</td>
<td>Bucket bucket = cluster.getBucket(&quot;users&quot;);</td>
</tr>
<tr>
<td>Connection conn = ds.getConnection();</td>
<td>// access data via a bucket</td>
</tr>
<tr>
<td>// access data via a connection</td>
<td></td>
</tr>
</tbody>
</table>

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

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 the users bucket, N1QL queries are not.

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.

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

Figure 7: Working with domain objects vs. document objects
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. However, transactions must be initiated in Java by the database client. This may require converting SQL based transactions (i.e. BEGIN TRANSACTION, COMMIT) to code.

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.

```java
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();
});
```
Couchbase also provides an asynchronous API. It is possible to use transactions asynchronously.

```java
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 9: Transaction using the Java asynchronous API*

**Installing and scaling your database**

One of the key advantages driving 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 leverage distribution to scale out - more data requires more disks, not a bigger one, and more users require more servers, not a bigger one.

*Couchbase Server leverages topology-aware clients and consistent hashing to distribute data within a cluster. In addition, data can be replicated to one or more nodes to provide high availability.*
NOTE:
NoSQL databases should be deployed as a cluster to realize all of their benefits.

NOTE:
You can complete the installation with the default configuration values, then change them later.

Installing 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.

Figure 11: Topology-aware clients in Couchbase Server

Figure 12: Couchbase Server configuration via the web admin console
Creating a cluster

After configuration is complete, the cluster overview page will be displayed.

Adding nodes to create a cluster is simple. During configuration of a new node, select “Join a cluster now.” instead of “Start a new cluster.” After joining the cluster, simply rebalance the data. While data is logically stored in a bucket, it is physically stored in vBuckets. A bucket is comprised of 1,024 vBuckets. The data is partitioned by distributing vBuckets to nodes within the cluster. When one or more nodes is added, some of the vBuckets are transferred from existing nodes to the new ones. This not only increases capacity - since every node stores less data, thereby making room for more data - it also evens the load by spreading it across more nodes.

Scaling beyond a single cluster

While many relational databases require separate software to replicate data between multiple data centers, some NoSQL databases (such as Couchbase Server) include cross datacenter replication (XDCR).

NOTE:
Couchbase Server is engineered with a memory-centric architecture. The more memory allocated, the better it should perform. Couchbase Server is deployed in production environments with up to 1TB of memory per node. However, the minimum is only 256MB.

While many relational databases require separate, complex configuration to replicate data between multiple datacenters – for example, Microsoft SQL Server requires failover clustering, always-on availability groups, or peer-to-peer transaction replication – some NoSQL databases (such as Couchbase Server) include cross datacenter replication (XDCR) built in for simplicity. However, not all NoSQL databases support bidirectional replication. In addition, many handle XDCR by deploying a single cluster across multiple datacenters and performing standard, intercluster replication between them. This not only increases operational complexity and overhead, it limits flexibility. In contrast, Couchbase Server supports both unidirectional and bidirectional replication with a dedicated XDCR protocol between small, independent clusters to simplify management and provide greater flexibility.
Monitoring and managing your deployment

While many relational and NoSQL databases require separate administration tools, Couchbase Server includes 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/views
- Cross datacenter 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:
- `cbbackup` and `cbrestore` – full, cumulative, and incremental backup and restore
- `cbcollect_info` and `cbdstats` – gather node and cluster diagnostics (280+ metrics)
- `cbq` – run N1QL queries from the command line
- `cbtransfer` – transfer data to and from clusters or CSV files
Putting it all together: How to conduct a successful proof of concept

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 solution 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. Try to find an application that 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 add nodes). It may be faster development cycles. Whatever it may be, make sure to specify it up front.

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, queries, or MapReduce views – 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 a production deployment. This is the time to review architecture and create a blueprint for production. Based on 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 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, while 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.

About Couchbase

Unlike other NoSQL databases, Couchbase provides an enterprise-class, multicloud to edge database that offers the robust capabilities required for business-critical applications on a highly scalable and available platform. As a distributed cloud-native database, Couchbase runs in modern dynamic environments and on any cloud, either customer-managed or fully managed as-a-service. Couchbase is built on open standards, combining the best of NoSQL with the power and familiarity of SQL, to simplify the transition from mainframe and relational databases.

Couchbase has become pervasive in our everyday lives; our customers include industry leaders Amadeus, American Express, Carrefour, Cisco, Comcast/Sky, Disney, eBay, LinkedIn, Marriott, Tesco, Tommy Hilfiger, United, Verizon, as well as hundreds of other household names. For more information, visit www.couchbase.com.

© 2020 Couchbase. All rights reserved.