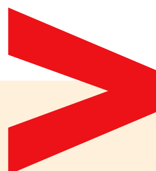
Couchbase

GUIDE

Data Modeling Guide

Contents

COMPARING DOCUMENT-ORIENTED AND RELATIONAL DATA	**3**
USING JSON DOCUMENTS	**6**
SCHEMALESS DATA MODELING	**8**
PHASES OF DATA MODELING	**10**
Logical Data Modeling	10
Analyze Your ***Logical*** Model	12
When to Embed and When to Refer	13
Physical Data Modeling	17
Documents	18
JSON DESIGN CHOICES	**19**
Document Key Prefixing	19
Document Management Fields	20
Field Name Length, Style, Consistency	21
Objects vs. Object Arrays	22
Array Element Complexity and Use	22
Time Series: A Specialized Array	23
Vector: A Specialized Array	24
Timestamp Format	26
Four States of Data Presence in JSON Docs	27
KEY DESIGN	**28**
Prefixing	29
Predictable	29
Counter ID	30
Unpredictable	31
Combinations	31
STANDARDIZED FIELDS	**32**
docType	32
Versioning	32
OPTIMIZATIONS	**33**
Optimizing Dates	33
RESOURCES	**34**



This document describes core elements you will use to model and handle data in Couchbase Capella™ (or Couchbase Server). It describes the ways you can structure individual JSON documents for your application, how to store the documents with a Couchbase SDK, and different approaches you may take when you structure data in documents.

Couchbase is a document database. Unlike traditional relational databases, you store information in JSON documents in collections rather than rows in tables. Documents generally contain all the information about a data entity (including compound data) rather than the data being normalized across tables.

A document is a JSON object consisting of a number of key-value pairs, or fields, that you define. There is no enforced schema in Couchbase; every JSON document can have its own individual set of fields, although you may probably adopt one or more informal schemas for your data.

Although Couchbase can store any kind of data, the benefit of storing JSON documents is that data can be indexed and queried beyond a simple key-value lookup.

COMPARING DOCUMENT-ORIENTED AND RELATIONAL DATA

In a relational database system you must define a ***schema*** before adding records to a database. The schema is the structure described in a formal language supported by the database and provides a blueprint for the tables in a database and the relationships between tables of data. Within a table, you need to define constraints in terms of rows and named columns as well as the type of data that can be stored in each column.

In contrast , a document-oriented database contains ***documents***, which are records that describe the data in the document, as well as the actual data. Documents can be as complex as you choose; you can use nested data to provide additional subcategories of information about your object. You can also use one or more documents to represent a real-world object. The following compares a conventional table with document-based objects:

Beers Table

1167	Ale C	Miller	570
3424	Beerio	lans	340
5612	Amstel	Amtel	121
2409	Colt's	BeerCo	98

Beer Documents

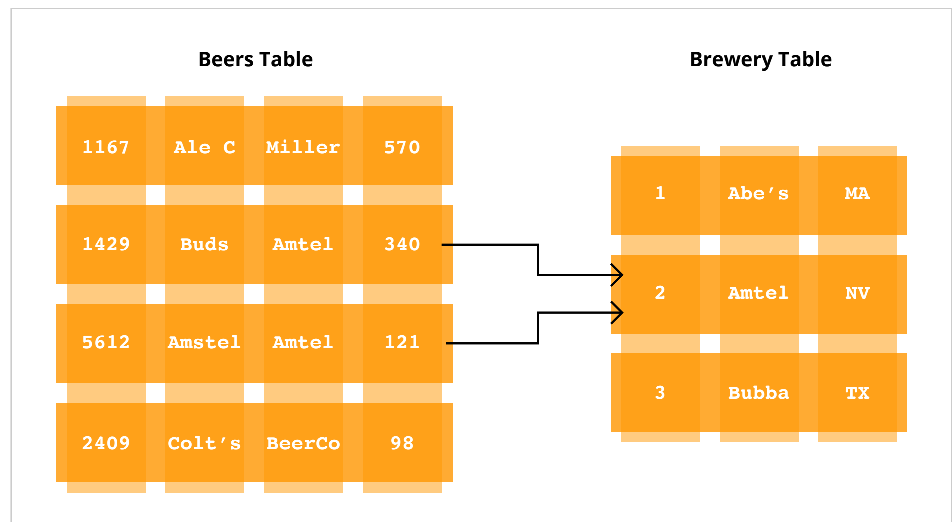
```
beer_1167

{_id: "1167",
name: "Ale C",
brewer: "Miller",
units: 570
}
```

```
24",
erio",
ans",
```

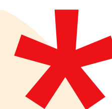
In this example we have a table that represents beers and their respective attributes: id, beer name, brewer, bottles available, and so forth. As we see in this illustration, the relational model conforms to a schema with a specified number of fields which represent a specific purpose and data type. The equivalent document-based model has an individual document per beer; each document contains the same types of information for a specific beer.

In a document-oriented model, data objects are stored as documents; each document stores your data and enables you to later update the data or delete it. Instead of columns with names and data types, we describe the data in the document, and provide the value for that description. If we wanted to add attributes to a beer in a relational model, we would need to modify the database schema to include the additional columns and their data types. In the case of document-based data, we would only need to add additional fields on a per-document basis.

Another characteristic of a relational database is ***data normalization***; this means you decompose data into smaller, related tables. The figure below illustrates this:

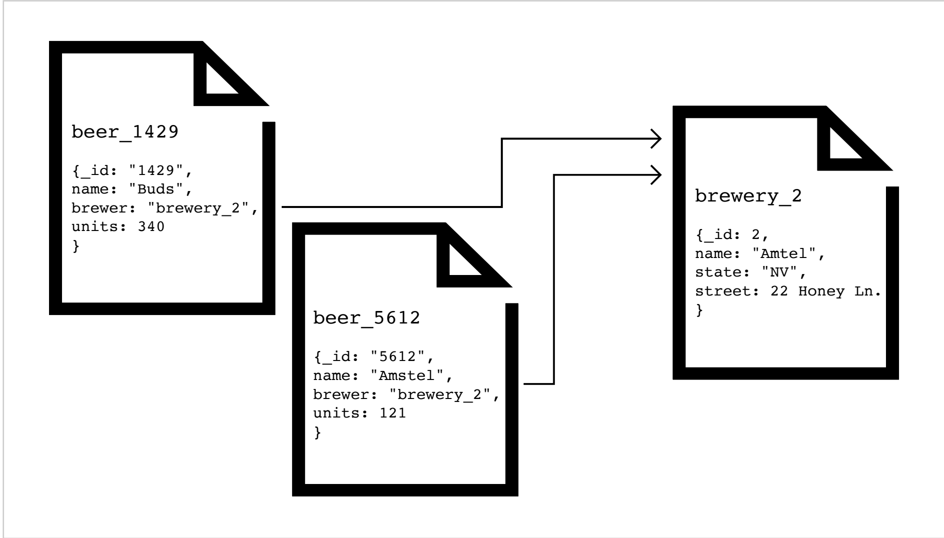


In the relational model, data is shared across multiple tables. The advantage to this model is that there is less duplicated data in the database. If we did not separate beers and breweries into different tables and had one beer table instead, we would have repeated information about breweries for each beer produced by that brewer.



One problem with this approach is that when you change information across tables, you need to lock those tables simultaneously to ensure information changes across the table consistently. Because you also spread information across a rigid structure, it makes it more difficult to change the structure during production, and it is also difficult to distribute the data across multiple servers.

In the document-oriented database, we could choose to have two different document structures: one for beers, and one for breweries. Instead of splitting your application objects into tables and rows, you would turn them into collections and documents. By providing a reference in the beer document to a brewery document, you create a relationship between the two entities:



In this example we have two different beers from the Amstel brewery. We represent each beer as a separate document and reference the brewery in the brewer field. The document-oriented approach provides several upsides compared to the traditional RDBMS model. First, because information is stored in documents, updating a schema is a matter of updating the documents for that type of object. This can be done with no system downtime. Second, we can distribute the information across multiple servers with greater ease. Since records are isolated and contained within entire documents, it makes it easier to move, or replicate, an entire object to another server.

USING JSON DOCUMENTS

JavaScript Object Notation (JSON) is a lightweight data-interchange format which is easy to read and change. JSON is language-independent (although it is derived from JavaScript). JSON documents enable you to benefit from all the Couchbase features, such as indexing and querying; they also provide a logical structure for more complex data and enable you to provide logical connections between different records.

The following are basic data types supported in JSON:

- Numbers, including integer and floating point
- Strings, including all Unicode characters and backslash escape characters
- Boolean, true or false
- Arrays, enclosed in square brackets: ["foo", "bar", "baz"]
- Objects, consisting of JSON field pairs, and also known as an ***associative array*** or hash. The name must be a string and the value can be any supported JSON data type.

For more information about creating valid JSON documents, please refer to https://www.json.org/.

When you use JSON documents to represent your application data, you should think about the document as a logical container for information. This involves thinking about how data from your application fits into natural groups. It also requires thinking about the information you want to manage in your application. Modeling data for Couchbase is similar to the process that you would do for traditional relational databases; there is, however, much more flexibility, and you can more easily change your mind later on your data structures. During your data/document design phase, you want to evaluate:

What are the ***things*** you want to manage in your applications? For instance, ***users***, ***breweries***, ***beers***, and so forth.

What do you want to store about the ***things***? For example, this could be ***alcohol percentage***, ***aroma***, ***location***, etc.

How do the ***things*** in your application fit into natural groups?

For instance, if you are creating a beer application, you might want a particular document structure to represent a beer:

```
{
  "name": "Hoptimus Prime",
  "description": "North American Ale Beer",
  "category": "North American Ale",
  "updated": "2010-07-22 20:00:20"
}
```

For each of the fields in this JSON document, you would provide unique values to represent individual beers. If you want to provide more detailed information in your beer application about the actual breweries, you could create a JSON structure to represent a brewery:

```
{
  "name": "Legacy Brewing Co.",
  "address": "525 Canal Street",
  "city”": "Reading",
  "state": "Pennsylvania",
  "website”": "legacybrewing.com",
  "description": "Brewing Company"
}
```

Performing data modeling for a document-based application is no different than the work you would need to do for a relational database. For the most part it can be much more flexible; it can provide a more realistic representation that closer matches the objects used in your application, and it also enables you to change your mind later about data structure.

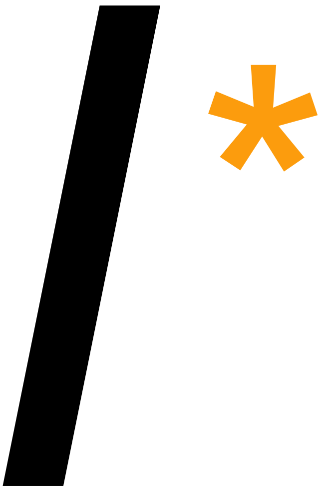
For more complex items in your application, one option is to use nested objects to represent the information:

```
{
  "name": "Legacy Brewing Co.",
  "address": "525 Canal Street",
  "city": "Reading",
  "state": "Pennsylvania",
  "website": "legacybrewing.com",
  "description": "Brewing Company",
  "geo": {
     "location": [
        "-105.07",
        "40.59"
     ],
     "accuracy": "RANGE_INTERPOLATED"
  },
  "beers": [
     "beer:id4058",
     "beer:id7628"
  ]
}
```

In this case, we added a nested attribute for the geolocation of the brewery and for beers. Within the location, we provide an exact longitude and latitude, as well as level of accuracy for plotting it on a map. The level of nesting you provide is your decision; as long as a document is under the maximum storage size for Couchbase, you can provide any level of nesting that you can handle in your application.

In traditional relational database modeling, you would create tables that contain a subset of information for an item. For instance a ***brewery*** may contain types of beers which are stored in a separate table and referenced by the ***beer id***. In the case of JSON documents, you use fields or even nested objects.

SCHEMALESS DATA MODELING



When you use documents to represent data, documenting a database schema is optional, and the majority of your effort will be creating one or more documents that will represent application data. This document structure can evolve over time as your application grows and adds new features.

In Couchbase, you do not need to perform data modeling and establish relationships between tables the way you would in a traditional relational database. Technically, every document you store with structure in Couchbase has its own *implicit* schema; the schema is represented in how you organize and nest information in your documents.

As a developer, you benefit in several ways from this approach:

- Extend the schema at runtime, or anytime. You can add new fields for a type of item anytime. Changes to your schema can be tracked by a version number, or by other fields as needed.
- Document-based data models may better represent the information you want to store and the data structures you need in your application.
- You design your application information in documents, rather than model your data for a database.
- Converting application information into JSON (serializing) is very simple; there are many options, and there are many libraries widely available in every platform and language.
- Minimization of one-to-many relationships through use of nested entities and therefore, reduction of joins (and the performance penalty they required).

There are several considerations to have in mind when you design your JSON document:

- What scopes and collections to create to store and organize your documents. There is no requirement to use anything but a single collection, but there are many organizational and indexing benefits.
- What particular keys, ids, prefixes, or conventions you want to use for items, for instance, `beer_My_Brew, My_Brew, f78a2ccd-6ce4-43bf-a6df-aa9dd1a40f7b, and 00001` are all valid choices with different benefits and trade-offs.
- When you want a document to expire (automatically deleted after a period of time), if at all, and what expiration would be best.
- If you want to use a document to access other documents: In other words, you can store keys that refer to other documents in a JSON document and get the keys through this document. In the NoSQL database jargon, this is often known as using ***composite keys***.

In Couchbase, **documents** are organized within collections. Collections are organized inside of scopes, and scopes are organized inside of buckets. Buckets are organized in a cluster.

For data modeling purposes, the main focus is **collections**. These are analogous to tables in the relational world, and help you to group like documents together. As stated previously, there is no requirement to do so, but it's a good idea to keep your data organized.

Scopes are less important for modeling, but can factor in, especially with a multi-tenant system, to restrict data to a microservice. These are analogous to a relational schema (e.g., in "dbo.Users", the schema is "dbo").

Buckets should not be used for modeling purposes, except in rare instances. Buckets each have their own replication settings and memory quotas. They are most closely analogous to a relational "database" or "catalog".

Finally, **clusters** are roughly equivalent to a relational "server," except data is distributed across multiple machines in a cluster. Data modeling at the cluster level is also rare, but in the case of modeling and gating data in different regions, it might be useful (e.g., U.S. cluster and EU cluster, with GDPR restrictions limiting the data that can sync between them).

What happens if we want to change the fields we store for a brewery? In this case we just add the fields to brewery documents. In this example we decide later that we want to include GPS location of the brewery:

```
Key: "brewery_Legacy_Brewing_Co", Document: {
  "name": "Legacy Brewing Co.",
  "address": "525 Canal Street, Reading, Pennsylvania, 19601
United States",
  "updated": "2010-07-22 20:00:20",
  "latitude": -75.928469,
  "longitude": 40.325725
}
```

(In Couchbase, the document key is not part of the JSON. It's part of a document's metadata. In this document, that will be represented like "Key: <key value>, Document: <json>".)

So in the case of document-based data, we can extend *just this individual record* by simply adding the two new fields for ***latitude*** and ***longitude***. When we add other breweries after this one, we could include these two new fields. For older breweries we can update them later with the new fields or provide programming logic that shows a default for older breweries.

A good idea when adding new fields to a document is to perform a compare and swap operation on the document to change it; with this type of operation, Couchbase will send you a message that the data has already changed if someone has already changed the record, and prevent conflicts or loss of data. For more information about compare and swap methods with Couchbase, see Compare and Swap (CAS).

Additionally, if you need to make updates to two related documentations, use an ACID transaction to prevent data inconsistency.

To create relationships between items, again we use fields. In the example below, we create a logical connection between beers and breweries using a ***brewery*** field in our beer document which relates to the ***key*** of the brewery document. This is analogous to the idea of using a foreign key in traditional relational database design.

This first document represents a beer, Hoptimus Prime:

```
Key: "beer_Hoptimus_Prime", Document: {
  "abv": 10,
  "brewery": "brewery_Legacy_Brewing_Co",
  "category": "North American Ale",
  "name": "Hoptimus Prime",
  "style": "Double India Pale Ale"
}
```

This second document represents the brewery that brews Hoptimus Prime:

```
Key: "brewery_Legacy_Brewing_Co", Document: {
  "name": "Legacy Brewing Co.",
  "address": "525 Canal Street Reading, Pennsylvania, 19601 
United States",
  "updated": "2010-07-22 20:00:20",
  "latitude": -75.928469,
  "longitude": 40.325725
}
```

In the beer document, the ***brewery*** field points to ***"brewery_Legacy_Brewery_Co"***, which is the key for the document that represents the brewery. By using this model of referencing documents within a document, we can create relationships between application objects.

PHASES OF DATA MODELING

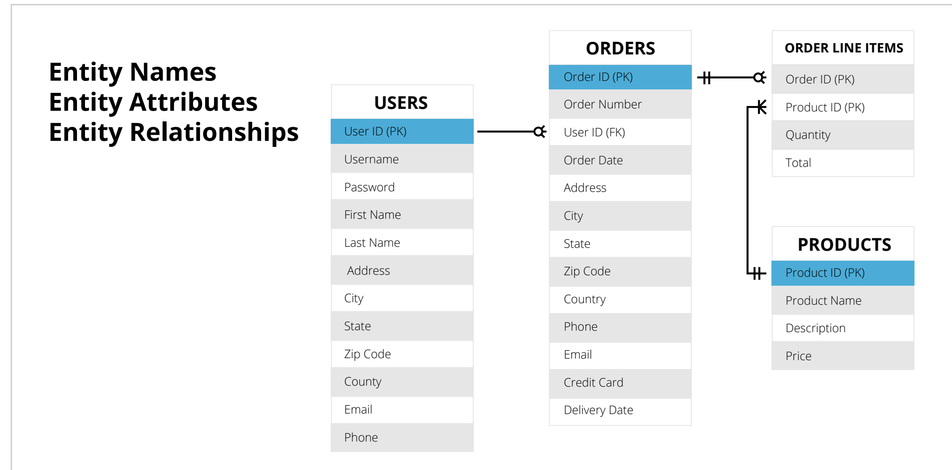
A data modeling exercise typically consists of at most three phases: **conceptual data modeling**, **logical data modeling**, and **physical data modeling**. (This document focuses on logical and physical, since the main focus of conceptual modeling involves domain expertise.) Logical data modeling focuses on describing your entities and relationships (typically this is done before even picking a database technology). Physical data modeling takes the logical data model and maps the entities and relationships to physical containers (that is, a database technology).

Logical Data Modeling

The logical data modeling phase focuses on describing your entities and relationships. Logical data modeling is done independently of the requirements and facilities of the underlying database platform.

At a high level, the outcome of this phase is a set of entities (objects) and their attributes that are central to your application's objectives, as well as a description of the relationships between these entities. For example, entities in an order management application might be ***users***, ***orders***, ***order items***, and ***products*** where their relationships might be "users can have many orders, and in turn each order can have many items."

Let's look at some of the main definitions you need from your logical data modeling exercise:



Entity names: Each group of entities has a name. In the above diagram, they are "users", "orders", "order line items", and "products". In a relational database, these will often become tables. In Couchbase, these will often become collections (or embedded within another document).

Entity keys: Each entity instance is identified by a unique key (in Couchbase, this is called a "document key" or "document ID"). The unique key can be a composite of multiple attributes or a surrogate key generated using a counter or a UUID generator. Composite or compound keys can be utilized to represent immutable properties and efficient processing without retrieving values. The key can be used to reference the entity instance from other entities for representing relationships.

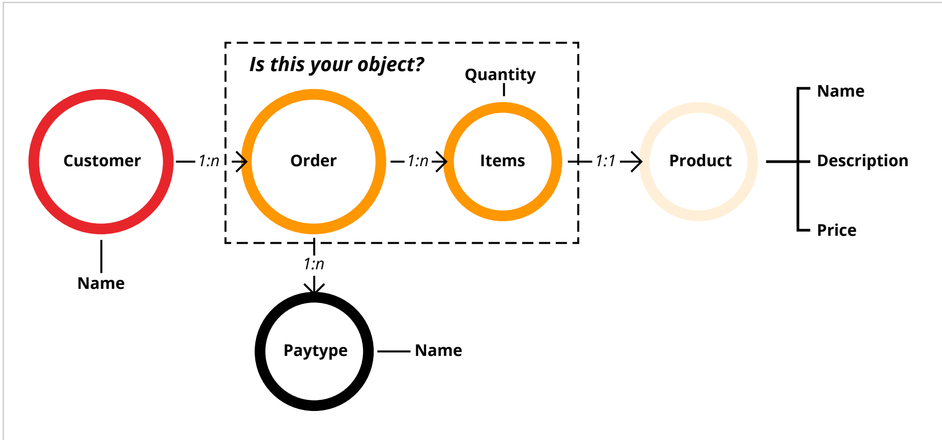
Entity attributes: Attributes can be any of the basic data types such as string, numeric, or Boolean, or they can be an array of these types. For example, an order might define a number of simple attributes such as ***order id*** and ***quantity***, as well as a complex attribute like ***product*** which in turn contains the attribute's ***product name***, ***description***, and ***price***.

Entity relationships: Entities can have 1-to-1, 1-to-many, or many-to-many relationships. For example, "an order has many items" is a 1-to-many relationship. There are many ways to diagram relationships, but the above diagram uses lines between the entities, with various symbols to indicate the type of relationship.

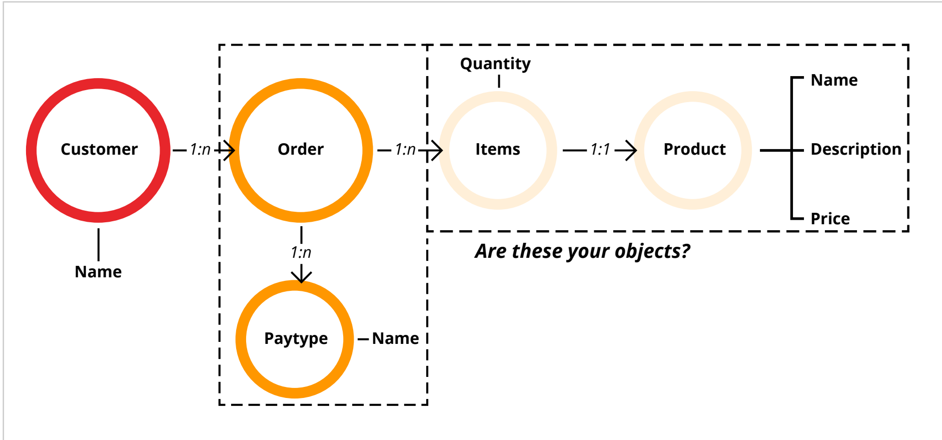
Analyze Your *Logical* Model

Let's look at a highly simplified order management system (OMS) as an example.

Modeling option 1: In the diagram below, Order embeds Items, and refers to external Product (1:n) and Paytype (1:1) docs.

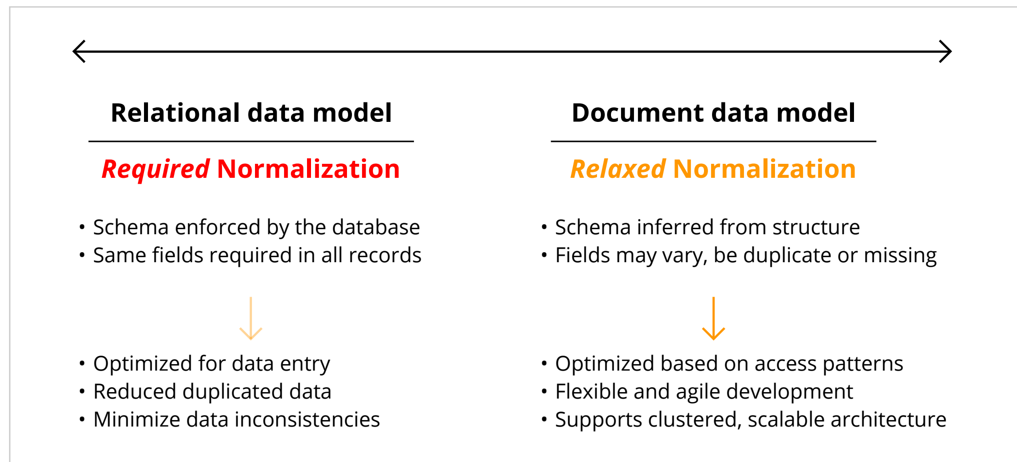


Modeling option 2: In the diagram below, Order embeds Paytype and refers to Items which embeds Product.



Logical data modeling starts with a decision on how to map your entities to documents. In the former example, an order document contains the items, but the paytype is a separate document. In the latter example, the items are separate from the order (but paytype is now a part of order). JSON documents provide great flexibility in mapping 1-to-1, 1-to-many, or many-to-many relationships.

In JSON data modeling, there is a spectrum of "no embedded" (i.e., relational style data) and "embedding everything."



At one end, you can model each entity to its own document with references to represent relationships. At the other end, you can embed all related entities into a single large document. However, the right design for your application usually lies somewhere in between. Exactly how you should balance these alternatives depends on the access patterns and requirements of your application.

When to Embed and When to Refer

In order to decide when to embed and when to refer, it can often be a good idea to consider the path of the data retrieval and manipulation.

Let's take a look at the example of a stock management system to track Couchbase-branded swag. Let's imagine the standard path is:

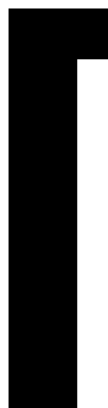
1. A customer makes an order.

2. A stock picker receives the order and packages the items.

3. A dispatcher sends out the package through a delivery service.

At the moment the customer makes an order, we have a choice of how we store the order data in Couchbase:

- Embed all the order information in one document; or
- Maintain one main copy of each record involved and refer to it from the order document.



EMBEDDING:

If we chose to embed all the data in one document, we might end up with something like this:

```
Key: "200", Document: {
  "customer": {
    "name": "Steve Rothery",
    "address": "11-21 Paul Street",
    "city": "London"
  },
  "products": [
    {
      "itemCode": "RedTShirt",
      "itemName": "Red Couchbase t-shirt",
      "supplier": "Lovely t-shirt company",
      "location": "warehouse 1, aisle 3, location 4",
      "quantityOrdered": 3
    },
    {
      "itemCode": "USB",
      "supplier": "Memory Sticks Foreva",
      "itemName": "Black 8GB USB stick with red Couchbase logo",
      "location": "warehouse 1, aisle 42, location 12",
      "quantityOrder": 51
    }
  ],
  "status": "paid"
}
```

Here, everything we need to fulfill the order is stored in one document. There are still separate customer profile and item details documents, but we replicate (copy) parts of their data in the order document. This might seem wasteful or even dangerous, if you're coming from the relational world. However, it's quite normal for a document database. As we saw earlier, document databases operate around the idea that one document could store everything you need for a particular situation.

There are, however, some trade-offs to embedding data like this.

First, let's look at what's potentially bad:

- ***Inconsistency:*** If ***Steve*** wants to update his address after the order is made but not shipped, we're relying on:
  - Our application code to be robust enough to find every instance of his address in the database and update it.
  - Nothing going wrong on the network, database side, or elsewhere that would prevent the update completing fully.
- ***Queryability:*** By making multiple copies of the same data, it could be harder to query on the data we replicate as we'll have to filter out all of the embedded copies.
- ***Size:*** You could end up with large documents consisting of lots of duplicated data.
- ***More documents:*** This isn't a major concern but it could have some impact, such as the size of your cached working set.

So, what benefits does embedding give us? Mostly, it gives us:

- ***Speed of access:*** Embedding everything in one document means we need just one database look-up.
- ***Potentially greater fault tolerance at read time:*** In a distributed database our referred documents would live on multiple machines, so by embedding we're introducing fewer opportunities for something to go wrong and we're simplifying the application side.

WHEN TO EMBED:

You might want to embed data when:

1. Reads greatly outnumber writes.
2. You're comfortable with the risk of inconsistent data across the multiple copies (i.e., if the copied data doesn't change that often).
3. You're optimizing for speed of access.

Why are we asking whether reads outnumber writes?

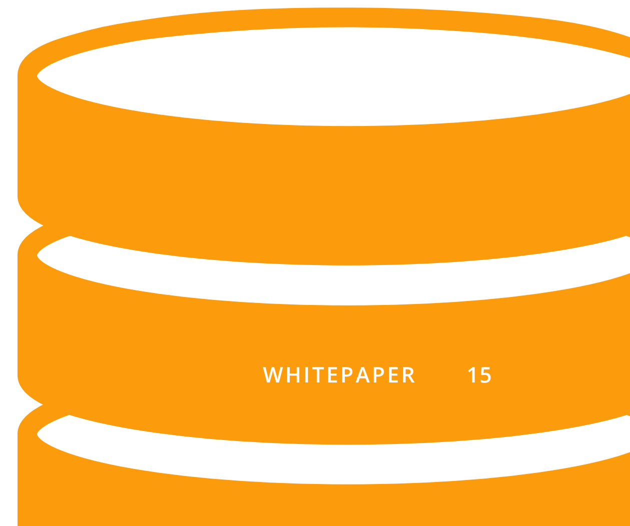
In our example above, each time someone reads our order they're also likely to update the state of the order:

- Someone in the warehouse reads the order document and updates the status to ***Picked***, once they're done.
- One of our dispatch team reads the document and updates the status to ***Dispatched*** when the parcel is with the courier.
- When we receive an automated delivery notice from the courier, our application updates the document status to ***Delivered***.

So, in this scenario, the reads and writes are likely to be fairly balanced.

Imagine, though, that we add a blog to our swag management system and then write a post about our new Couchbase-branded USB charger. We'd make two, maybe three, writes to the document while finishing our post. Then, for the rest of that document's lifetime, it'd be all reads. If the post is popular, we could see a hundred or thousand times the number of reads compared to writes.

As the benefits of embedding come at read-time, and the risks mostly at write-time, it seems reasonable to embed all the contents of the blog post page in one document rather than, for example, pull in the author details from a separate profile document.



There's another compelling reason to embed data:

> You actually *want* to maintain separate, and divergent, copies of data.

In our swag order above, we're using the customer's address as the dispatch address. By embedding the dispatch address, as we are, we can easily offer the option to choose a different dispatch address for each order. We also get a historic record of where each order went even if the customer later changes the address stored in their profile.

REFERRING:

Another way to represent our order would be to refer to the user profile document and stock item details document but not to pull their contents into the order document.

Let's imagine our customer profiles are keyed by the customer's email address and our stock items are keyed by a stock code. We can use those to refer to the original documents:

```
Key: "200", Document: {
  "customer": "steve@gmail.com",
  "products": [
     {
        "itemCode": "RedTShirt",
        "quantityOrdered": 3
     },
     {
        "itemCode": "USB",
        "quantityOrder": 51
     }
  ],
  "status": "paid"
}
```

When we view ***Steve's*** order, we can fill in the details with two more reads: his user profile (keyed by the email address) and the stock item details (keyed by the item codes).

It requires two additional reads but it gives us some benefits:

- ***Consistency:*** We're maintaining one canonical copy of Steve's profile information and the stock item details.
- ***Queryability:*** This time, when we query the dataset we can be more sure that the results are the canonical versions of the data rather than embedded copies.
- ***Better cache usage:*** As we're accessing the canonical documents frequently, they'll stay in our cache, rather than having multiple embedding copies that are accessed less frequently and so drop out of the cache.
- ***More efficient hardware usage:*** Embedding data gives us larger documents with multiple copies of the same data; referring helps reduce the disk and RAM our database needs.

There are also disadvantages:

- ***Multiple look-ups:*** This is mostly a consideration for cache misses, as the read time increases whenever we need to read from disk.
- ***Consistency is enforced:*** Referring to a canonical version of a document means updates to that document will be reflected in every context where it is used.

WHEN TO REFER:
Except for satellite data (like “order items” embedding into an order document), referring to canonical instances of documents is a good default when modeling with Couchbase. You should be especially keen to use referrals when:

- Consistency of the data is a priority.
- You want to ensure your cache is used efficiently.
- The embedded version would be unwieldy.

That last point is particularly important where your documents have an unbound potential for growth.

Imagine we were storing activity logs related to each user of our system. Embedding those logs in the user profile could lead to a rather large document.

It’s unlikely we’d breach Couchbase’s 20MB upper limit for an individual document, but processing the document on the application side could be less efficient as the log element of the profile grows. It’d be much more efficient to refer to a separate document, or perhaps paginated documents, holding the logs.

General Guidelines on Nest/Refer

If…	Then consider…
Relationship is 1:1 or 1:many	Nest related data as nested objects
Relationship is many:1 or many:many	Refer to related data as separate docs
Reads are mostly parent fields	Refer to children as separate docs
Reads are mostly parent+child fields	Nest children as nested objects
Writes are mostly either parent or child	Refer to children as separate docs
Writes are mostly both parent and child	Nest children as nested objects

Physical Data Modeling

The physical data model takes the logical data model and maps the entities and relationships to physical containers.

In Couchbase, items are used to store associated values that can be accessed with a unique key. Couchbase also provides buckets to group items. Based on the access patterns, performance requirements, and atomicity and consistency requirements, you can choose the type of container(s) to use to represent your logical data model.

The data representation and containment in Couchbase is different from relational databases, but there are some familiar concepts that can help ease your transition. The following table provides a high-level comparison to help you get familiar with Couchbase containers.

Couchbase	Relational databases
Cluster	Server
Buckets	Database/Catalog
Scopes	Schema (e.g., "dbo")
Collections	Tables
Documents	Rows
Document Key/ID	Primary Key
SQL++ Index	SQL Index

Documents

Documents consist of a key and a value. A key is a unique identifier within a collection. A document's value can be a binary or a JSON document. You can mix binary and JSON values inside a collection.

Keys: Each value (binary or JSON) is identified by a unique key. The key is often a "surrogate" key generated using a counter or a UUID generator, since keys are immutable. Compound/composite keys are application-managed (e.g., "foo::bar", "baz::qux::1", etc.). So, if you use composite or compound keys, ensure that you use attributes that don't change over time.

Binary values: Binary values can be used for high-performance access to compact data through keys. Encrypted secrets, IoT instrument measurements, session states, or other non-human readable data are typical cases for binary data. ***Binary*** data may not necessarily be binary, but could be non-JSON formatted text like XML, string, etc. However, using binary values limits the functionality your application can take advantage of, ruling out indexing and querying in Couchbase, as binary values have a proprietary representation.

JSON values: JSON provides rich representation for entities. Couchbase can parse, index, and query JSON values. JSON provides a name and a value for each field. You can find the JSON definition at RFC 7159 or at ECMA 404.

The JSON document fields can represent both basic types such as number, string, Boolean, and complex types including embedded documents and arrays. In the examples below, a1 and a2 represent attributes that have a numeric and string value respectively, a3 represents an embedded document, and a4 represents an array of embedded documents.

```
{
  "a1": 123,
  "a2":"string",
  "a3":{
     "b1":[ 45, 67, 89]
  },
  "a4":[
     { "c1":"string", "c2":12 },
     { "c1":"string", "c2":34 }
  ]
}
```

JSON DESIGN CHOICES

Couchbase neither enforces nor validates for any particular document structure: as long as the JSON is valid and the key is unique, no further validation is enforced. Below are the design choices that impact JSON document design.

Document typing and versioning

- Key prefixing
- Document management fields

Document structure choices

- Field name choice, length, style, consistency, etc.
- Use of root attribute
- Objects vs. arrays
- Array element complexity
- Timestamp format
- Valued, missing, empty, and null attribute values

Document Key Prefixing

The document ID is the primary identifier of a document in the database. However, you may want to prefix keys for readability, export, backup, etc. For instance, if you have a users collection, documents within the users collection may contain keys of the form “user::123”. However, this is not required. You may want to avoid this, since each key will have six additional characters (in this case) that will take up space in caching RAM.

However, when exporting data to CSV or raw JSON files, you may want this information. Another approach would be to use a “type” or “docType” or other fields within the JSON itself. Some application frameworks like Spring Data will do this automatically anyway. And, if you plan to use two different types within a single collection (not recommended, but may be appropriate in some situations), a key prefix or type field may be necessary to differentiate.

Use an approach that works best for your team or use case, but keep in mind the trade-offs of these approaches:

Approach 1: A user document with no key prefix, no type data, stored in a user collection. This is the most efficient footprint, but is missing information that application frameworks (like Spring) might need.

```
Key: "123", Document: {
  "name": "Matt",
}
```

Approach 2: A user document with a key prefix and type data. This is the most descriptive footprint, but it also contains the most overhead.

```
Key: "user::123", Document: {
  "name": "Matt",
  "type": "user"
}
```

Of course, you can mix and match, change the delimiter (“::” is not required, you can use anything), change the field names (“type” is not a reserved word or anything, you can use “docType”, “class”, etc.), or any combination of the above as makes sense for your team.

Document Management Fields

A JSON document should also contain a version property to help manage changes in data structure. Depending on your application requirements, use case, the line of business, etc., other common properties to consider:

- **_created** – A timestamp of when the document was created (in epoch time, milliseconds or seconds, if millisecond precision is not required)
- **_updated** – Timestamp of when updated
- **_createdBy** – A user ID/name of the person or application that created the document
- **_modified** – A timestamp of when the document was last modified in epoch time (milliseconds or seconds, if millisecond precision is not required)
- **_updatedBy** – A user who updated
- **_modifiedBy** – A user ID/name of the person or application that modified the document
- **_accessed** – A timestamp of when the document was last accessed in epoch time (milliseconds or seconds, if millisecond precision is not required)
- **_geo** – A 2 character ISO code of a country

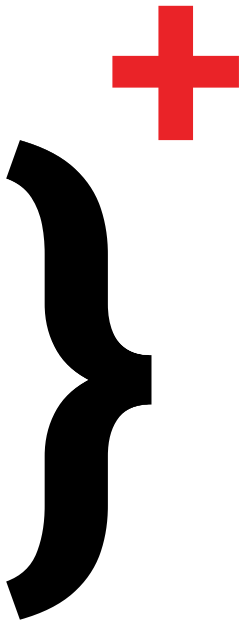
While not required, the use of a leading _ creates a standardized approach that conveys the purpose of these kinds of attributes across all documents within the enterprise.

Examples:

```
Key: "123", Document: {
  "name": "Matt",
  "_schema": "1.2",
  "_created": 1544734688923
}
```

These attributes can be grouped through a top-level property, i.e., "meta": {} .

```
{
  "meta": {
     "type": "user",
     "schema": "1.2",
     "created": 1544734688923
  },
  "name": "Matt",
}
```



Choose an approach that works within your organization and be consistent throughout your applications

Field Name Length, Style, Consistency

Some more tips to consider when naming fields:

- Brevity is beautiful at scale (e.g., a field name length of 11 vs. 6 characters may not seem like much, but multiply that times 1 billion documents and it adds up). Example: `geoCode vs countryCode`.
- Self-documenting names reduce doc effort/maintenance `userName vs usyslogintxt`.
- Consistent naming style reduces bugs and makes (de)serialization straightforward (PascalCase, camelCase, or snake_case are all fine, as long as you stick to a standard).
- Use plural names for array fields, and singular for others `"phones": [ ... ], "address": { ... }, "genre": " ... ", "scale": 2.3`.
- Avoid words that are reserved in SQL++ to avoid the need for escaping as much as possible: `user, bucket, cluster, role, select, insert`, etc. Please refer to SQL++ Reserved Words for more details and how to escape reserved words in SQL++.
- Beware of hyphens, as these will require escaping. Example: `first_name vs first-name`.

Objects vs. Object Arrays

There are two different ways to represent objects.

Objects – In this choice, `phones` is a nested object in a `userProfile` document. It can have named properties that correspond to different types of phone numbers:

```
{
   SBL_20240329_061  Workflows for Events_created": "2015-01-
28T13:50:56",
   "firstName": "Andy",
   "lastName": "Bowman",
   "phones": {
        "cell": "212-771-1834",
        "business: "212-770-4318"
   },
   "status": "active",
   "_updated": "2015-08-25T10:29:16"
}
```

Object arrays – In this choice, `phones` is an array of objects in the `userProfile` document:

```
{
   "_created": "2015-01-28T13:50:56",
   "firstName": "Andy",
   "lastName": "Bowman",
   "phones": [
        {
               "number": "212-771-1834",
               "type": "cell"
        }
   ],
   "status": "active",
   "_updated": "2015-08-25T10:29:16"
}
```

Array Element Complexity and Use

Array values may be ***simple*** or ***object***.

One common use of an array of simple values is to **store key(s) to lookup/join**:

With this approach, ***tracks*** is an array of strings which contain track ids. Let's say we have to get the track and ***artist*** name for each of the track id: this requires multiple ***gets***. So, this choice will have a significant impact when the user base is high. For instance, if you have 1M users, accessing all this information would translate to 3M ***gets***.

```
{
"_created": "2014-12-04T03:36:18",
"name": "Eclectic Summer Mix",
"owner": "copilotmarks61569",
"tracks": [
   "9FFAF88C1C3550245A19CE3BD91D3DC0BE616778",
   "3305311F4A0FAAFEABD001D324906748B18FB24A",
   "0EB4939F29669774A19B276E60F0E7B47E7EAF58"
],
"_updated": "2015-09-11T10:39:40"
}
```

Alternatively, you can nest a copy or summary to avoid a lookup/join. With this approach, all we have to do is ***one*** get to retrieve all the information that we need regarding the playlist.

```
{
"_created": "2014-12-04T03:36:18",
"name": "Eclectic Summer Mix",
"owner": "copilotmarks61569",
"tracks": [
  {
     "id": "9FFAF88C1C3550245A19CE3BD91D3DC0BE616778",
     "title": "Buddha Nature",
     "artist": "Deuter",
     "genre": "Experimental Electronic"
     },
  {
     "id": "3305311F4A0FAAFEABD001D324906748B18FB24A",
     "title": "Bluebird Canyon Stomp",
     "artist": "Beaver & Krause",
     "genre": "Experimental Electronic"
  }
],
"_updated": "2015-09-11T10:39:40"
}
```

Of course, there are trade-offs to this approach, as discussed earlier.

Time Series: A Specialized Array

Couchbase is a JSON document database, and is not, strictly speaking, a time series database. However, JSON is a flexible format, and with a specially formatted array, Couchbase can be used to store and query time series data.

In order to use all of Couchbase's time series functionality, a document must contain the following fields:

- ts_start – start date of the time series (epoch milliseconds)
- ts_end – end date of the time series (epoch milliseconds)
- ts_interval – interval between data points (in milliseconds)
- ts_data – an array (or an array of arrays) of data points

(Documents can contain other fields, too, but these are necessary for time series data).

Here's an example of 10 points of stock ticker data, starting on March 2, 2023 4:22:10 a.m., ending at 4:22:19 a.m., with an interval of 1000 ms between each data point:

```
{
"ticker": "BASE",
"ts_start": 1677730930000,
"ts_end": 1677730939000,
"ts_interval": 1000,
"ts_data": [ 16.30, 16.31, 16.32, 16.33, 16.34, 16.35, 16.36,
16.37, 16.38, 16.39 ]
}
```

Time series data can be queried like any other JSON in Couchbase. However, there's also a _TIMESERIES function built into SQL++ that will calculate the date/time of each point, and return expanded individual values. Here's an example containing time series data over 10 data points, using the _TIMESERIES function to return data from part of that range, expanded to individual data points:

```
WITH stocks AS (
  [ {
    "ticker": "BASE",
    "ts_start": 1677730930000,
    "ts_end": 1677730939000,
    "ts_interval": 1000,
    "ts_data": [ 16.30, 16.31, 16.32, 16.33, 16.34,
 16.35, 16.36, 16.37, 16.38, 16.39 ]
    },     {
    "ticker": "MSFT",
    "ts_start": 1677730930000,
    "ts_end": 1677730939000,
    "ts_interval": 1000,
    "ts_data": [ 416.30, 416.31, 416.32, 416.33, 416.34,
 416.35, 416.36, 416.37, 416.38, 416.39 ]
    }      ]
),
range_start AS (1677730930000),
range_end AS (1677730932000)
SELECT s.ticker, t.*
FROM stocks AS s
UNNEST _TIMESERIES(s, {"ts_ranges": [range_start, range_end]})
AS t
```

The result of that query:

```
[
  { "_t": 1677730930000, "_v0": 16.3, "ticker": "BASE" },
  { "_t": 1677730931000, "_v0": 16.31, "ticker": "BASE" },
  { "_t": 1677730932000, "_v0": 16.32, "ticker": "BASE" },
  { "_t": 1677730930000, "_v0": 416.3, "ticker": "MSFT" },
  { "_t": 1677730931000, "_v0": 416.31, "ticker": "MSFT" },
  { "_t": 1677730932000, "_v0": 416.32, "ticker": "MSFT" }
]
```

This formatting allows time series data to be stored in a dense format, while still allowing for a full expansion of individual data points.

Vector: A Specialized Array

Similar to time series, note that Couchbase is not a single-purpose vector database. But again, the JSON format is flexible enough to store vectors. And once again, additional functionality in Couchbase (in concert with models like OpenAI and/or LangChain, LlamaIndex, etc.), allows vector data to be queried for use cases like similarity search, retrieval-augmented generation (RAG), recommendation systems, etc.

A vector is a mathematical concept that defines a point in N-dimensional space. For example, [1.8, -5.04, 3.998] is an example of a three-dimensional vector. An embedding is a special type of vector that encodes semantic meaning based on an AI model. An OpenAI embedding from the "text-embedding-ada-002" model, for instance, produces 1536-dimensional vectors.

To get an embedding, you'll start by giving some data to an AI model. For instance, you can submit text to a text model:

POST https://api.openai.com/v1/embeddings

```
Body:
{
    "input": "A cool Couchbase T-shirt with a bold logo and a
wave design that shows off the power of data.",
    "model": "text-embedding-ada-002"
}


Response:
{
    "object": "list",
    "data": [
      {
          "object": "embedding",
          "index": 0,
          "embedding": [-0.022023635,-0.0041531054,... etc ...]
      }
    ],
    "model": "text-embedding-ada-002",
    "usage": {
      "prompt_tokens": 22,
      "total_tokens": 22
    }
}
```

The "embedding" field in the response is an array of 1536 numbers: the length of a vector for the "text-embedding-ada-002" model. This array can be stored alongside the data within a document in Couchbase:

```
{
    "name" : "tshirt",
    "description" : "A cool Couchbase T-shirt with a bold logo
and a wave design that shows off the power of data.",
    "description_embed_model" : "text-embedding-ada-002",
    "description_embed" : [ -0.022023635, -0.0041531054, ... etc
... ]
}
```

To perform a vector search on documents like these stored in Couchbase, create a full-text search index, adding "description_embed" as a vector field.

Now that you have a vector search index, you can find items that are semantically similar. For instance, let's say a user wanted to find "**clothing with a wavy design**". Textually, that search string would be unlikely to match the above T-shirt. A human knows that T-shirt is a type of clothing, but computers wouldn't be able to understand that: except with the aid of a large language model (LLM) embedding. The embedding codes the semantics into numerical values (an array of numbers), which can then be used to find other items that are semantically close.

"knn" stands for "k-nearest neighbor," which is an algorithm that finds the closest vectors to a given vector. In this example, k = 2, so it will find the two closest vectors in your Couchbase data to the given vector.

The embedding for **"clothing with a wavy design"** with the same model is: [-0.030775987, -0.0041826647, . . . etc . . .]. There are multiple ways to execute a vector search query (SDKs, REST API, UI). Here's an example of a search body to execute the search in a REST request:

```
{
  "fields": [ "*" ],
  "query": { "match_none": "" },
  "knn": [
    {
      "k": 2,
      "field": "description_embed",
      "vector": [       -0.030775987, -0.0041826647, ...etc... ]
  }
  ]
}
```

See the full vector search documentation for a more complete picture. Some important notes that may affect modeling:

- Couchbase can store any number of vectors in a document. You can have a vector for each field value you want to search, or you can combine field values into one lump value and store a single embedding – the choice is yours.
- It may be a good practice to store the name of the model alongside the embedding. New models are being created all the time, but you must use the same model consistently to create embeddings, so this may help you keep track.
- Vector search in Couchbase can be combined with SQL++ queries, as well as non-vector full-text search capabilities, geospatial, time series, etc. This is known as "hybrid search."

Timestamp Format

Famously, JSON does not have a time/date/timestamp/datetime/etc. type. Dates are commonly stored as either strings or numbers in JSON.

The following are examples of commonly used date formats:

ISO8601 – Human readable, and can be sorted lexicographically.

```
{
  "countryCode": "US",
  "gdp": 53548,
  "name": "United States of America",
  "region": "Americas",
  "region-number": 21,
  "sub-region": "Northern America",
  "_updated": "2010-07-15T15:34:27"
}
```

Time component array – This format can be useful when you're trying to group data. Lets say you want to generate a graphic visualization, this choice may be the best suited.

```
 {
   "countryCode": "US",
   "gdp": 53548,
   "name": "United States of America",
   "region": "Americas",
   "region-number": 21,
   "sub-region": "Northern America",
   "_updated": [ 2010, 7, 15, 15, 34, 27 ]
}
```

Epoch/Unix – Epoch format is a numeric value specifying the number of seconds that have elapsed since 00:00:00 Thursday, 1 January 1970. Epoch format is the most efficient in terms of brevity, especially if you reduce the granularity. It's also a very efficient format when you have to do some kind of date comparison, sorting etc., but it's not as efficient for a human to read.

```
 {
   "countryCode": "US",
   "gdp": 53548,
   "name": "United States of America",
   "region": "Americas",
   "region-number": 21,
   "sub-region": "Northern America",
   "_updated": 1279208067000
}
```

Whatever format you store date/time in, Couchbase's SQL++ has a wide variety of date functions to perform comparison, math, conversion, and parsing.

Four States of Data Presence in JSON Docs

It is important to understand that JSON supports optional properties (i.e., a property doesn't exist, as opposed to existing with a null or empty value). If a property has a null value, consider dropping it from the JSON unless there's a good reason not to. SQL++ makes it easy to identify missing or null property values. Be sure your application code handles the case where a property value is missing. Here are examples of queries and the kind of document the query would match:

Query for documents where a field has a value:

```
SELECT geocode FROM . . . WHERE geocode IS VALUED
```

```
{
  "geocode": "USA"
}
```

Query for documents where a field doesn't have a value:

```
SELECT geocode FROM . . . WHERE geocode IS NOT VALUED
```

```
{
  "geocode": ""
}
```

Query for documents where a field is missing:

```
SELECT geocode FROM . . . WHERE geocode IS [NOT] MISSING
```

```
{
}
```

Query for documents where a field is explicitly null:

```
SELECT geocode FROM . . . WHERE geocode IS [NOT] NULL
```

```
{
  "geocode": null
}
```

KEY DESIGN

A very important part of NoSQL database modeling is deciding how to design your document keys. This will often inform and be informed by how you plan to access your data from your application. Here are five common patterns when it comes to designing a key (there may be others, and combinations are also possible):

Prefixing

Predictable

Counter ID

Unpredictable

Combinations

Prefixing

As mentioned earlier, prefixing keys is not required. Like documents can exist in the context of a collection. However, as also discussed earlier, you may still want to do so. For both the sake of prefixing and combination keys, it's a good idea to pick a delimiter and use it throughout all of your enterprise. A common delimiter is "::", since it's a character combination that's not likely to be anything but a delimiter. But also keep in mind the earlier discussion of brevity. Examples of key design and key values:

DocType:ID `userprofile:fredsmith79 playlist:003c6f65-641a-4c9a-8e5e-41c947086cae`

AppName:DocType:ID `couchmusic:userprofile:fredsmith79`

DocType:ParentID:ChildID `playlist:fredsmith79:003c6f65-641a-4c9a-8e5e-41c947086cae`

Predictable

Making keys predictable can also improve performance and the ease of looking up documents by key. Let's say we're storing a user profile. Assuming no cookies, what are we guaranteed to know about our user after they've logged in? Their login name, at least.

So, if we want to make life easy for ourselves in retrieving our user profile, the login name can be used as a key, and then it's one lookup to get a user document. From that point on, everything else we need to know about that person could be derived from their user profile, in one way or another.

As mentioned earlier, keys are immutable, so this could prevent a user from easily changing their login name. To handle this, we now have to either create a new user profile under a new key (copy the profile over, and remove the old document) or create a look-up document (which would then require a second lookup upon login). We could insist that our users can never change their login names (and perhaps you've encountered applications in the wild that do this) but it's unreasonable to make our users suffer unnecessarily.

The main downside of a predictable key is that, usually, it'll be an element of the data that we're storing.

Key: user:john.doe@mail.com

```
{} JSON
  user_id : 123
  username : "jdoe"
  first_name : "John"
  last_name : "Doe"
  email : "john.doe@mail.com"
  password : "881421883cba2b527fdbbc50a94"
```

Key: user:jdoe

```
{} JSON
  user_id : 123
  username : "jdoe"
  first_name : "John"
  last_name : "Doe"
  email : "john.doe@mail.com"
  password : "881421883cba2b527fdbbc50a94"
```

Counter ID

In relational databases, primary keys are often integers that are generated or auto-incremented. We can get Couchbase to generate the key for us using a counter document. If you're using a counter ID pattern, every insert (not update) requires two mutations: One to increment the counter and the other to create the document using that counter number as the key.

Here's how it works:

1. Someone fills out the new user account form and clicks "Submit."
2. We increment a counter document (you can use one global counter, or a counter per collection, or some variation in between, depending on how unique you want document keys to be) and it returns the next number up (e.g., 123).
3. We create a new user profile document keyed with 123.
4. At this point, we can create additional look-up documents for things such as their user id, enabling us to do a simple look-up on the data we hold at login time.

Key: user:123

```
{} JSON
  user_id : 123
  username : "jdoe"
  first_name : "John"
  last_name : "Doe"
  email : "john.doe@mail.com"
  password : "881421883cba2b527fdbbc50a94"
```

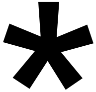
Key: user:counter

```
{} JSON
  Counter : 123
```

We also get some additional benefits from this pattern, such as a counter possibly providing us with some details of many user profiles we've created during the application's lifetime.

Unpredictable

This pattern uses system-generated unique ids like UUID. These keys are globally unique, and can be generated by an application without consulting Couchbase (unlike the counter method, which requires and additional operation).



Key: user:23ad6bac-7599-4874-af98-7af734027

```
{} JSON
  user_id : 23ad6bac-7599-4874-af98-7af734027
  username : "jdoe"
  first_name : "John"
  last_name : "Doe"
  email : "john.doe@mail.com"
  password : "881421883cba2b527fdbbc50a94"
```

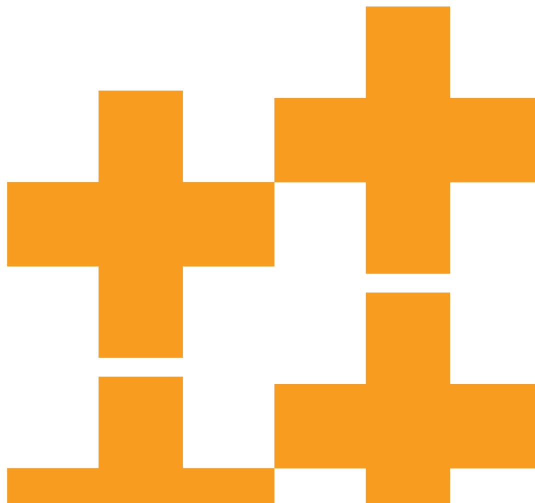
Combinations

Combining methods can also provide some interesting benefits.

We've looked before at when to embed data in one large document and when it's best to refer to other documents. When we choose to refer to data held in separate documents, we can build predictable key names from components that tell us something about what the document holds.

Let's look at our user profile example again. The main document is stored under the key 1001. We're working on an e-commerce site so we also want to know all of the orders our customer has made. Simple: we can store the entire list of orders in a single document with the key 1001::orders. Once logged in, it becomes a single key lookup to get the entire list of orders.

Similarly, our system might judge what sort of marketing emails to send to customers based on their total spend with the site. Rather than have the system calculate that each time, we can instead pre-calculate it and store it for later retrieval under the key 1001::orders::value.



STANDARDIZED FIELDS

docType

We discussed docType in an earlier section above. Please refer to Document Key Prefixing section in this document for details on docType.

Versioning

Applications are typically versioned using semantic versioning, i.e., 2.5.1. Where:

2 is the major version

5 is the minor version

1 is bugfix/maintenance version

Versioning the application informs users of features, functionality, updates, etc. The term "schemaless" is often associated with NoSQL, and while this is technically correct, it is better stated as:

> Note: There is no schema managed by the database, however, there is still a schema, and it is an "Application Enforced Schema." The application is now responsible for enforcing the schema as well as maintaining the integrity of the data and relationships.

As schemas change and evolve, documenting the version of the schema provides a mechanism of notifying applications about the schema version of the document that they're working with. This also enables a migration path for updating models which is discussed further in the Schema Versioning section.

```
{
  "name": "Matt Groves",
  "_schema": "1.2"
}

{
  "firstName": "Matt",
  "lastName": "Groves",
  "_schema": "2.0"
}
```

Please refer to the ***Document Management Strategies*** document for a more thorough discussion of schema versioning.

OPTIMIZATIONS

JSON gives us a flexible schema that allows our models to rapidly adapt to change. This is because the schema is explicitly stored alongside each value. Whereas, in an RDBMS, the schema is defined by the table columns, which are defined once. In any database, every byte of stored data adds up. With an application enforced schema, the model size is now controlled by the application. As developers, we can be verbose when describing variables throughout our applications, in order to make code easier to read. This practice tends to carry over to our JSON models. While it is generally preferred to maintain human-readable field names for developer productivity, there are often well-understood abbreviations for many fields that will not reduce document readability.

If your team is at the point where your model is documented and understood by everyone, your application is functional, and you have a handle on everything else covered in this document, here are some tips to reduce document size and help squeeze out better performance (by increasing available RAM for caching):

- Don't store the document ID as a repeated value in the document. This is redundant, and it risks getting out of sync with the key.
- Convert ISO-8601 timestamps to epoch time in milliseconds, saving at least 11 bytes for every date field. When millisecond precision is not required, convert to a smaller value (i.e., divide by 1,000 to convert to seconds, 60 for minutes, 60 for hours, 24 for days), saving at least 4 bytes.
- Store dates in ISO format YYYY-MM-DD instead of MM DD, YYYY.
- When using UUIDs, strip all dashes, saving an additional 4 bytes per UUID.
- Use shorter property names if possible.
- Don't store properties with null values, empty string/array/object, or a known default. Don't repeat values in arrays whose value is not unique; use a top-level property on the document instead.

Optimizing Dates

It is very common in almost any application, there is a need to store a date. This could be when the document was created, modified, when an order was placed, etc. Let's consider the ISO-8601 format.

Take the date `2018-12-14T03:45:24.478Z` as an example: this is very ***readable***, but is it the most efficient way to store the date? Storing this same date as Unix Epoch Time we can represent this same date as `1544759124478`. ISO-8601 is 24 bytes, where epoch format is 13 bytes, which saves 11 bytes. This might not seem like a lot, but consider this scenario: 500,000,000 documents and each document has an average of two date properties. If we used epoch format, we'd save 11,000,000,000 bytes or 11Gb of space.

Now, take this a step further and ask the question, "What level of precision does the application require?". Oftentimes we do not need millisecond precision; we can divide the epoch date accordingly for seconds, minutes, hours, etc. This applies if dates are being stored in Epoch format.

Epoch Date	Precision	Reduction	Output	Length/Bytes
1544759124478	milliseconds	n/a	1544759124478	13
1544759124478	seconds	/ 1000	1544759124	10
1544759124478	minutes	/1000 / 60	25745985	8
1544759124478	hours	/1000 / 60 / 60	429099	6
1544759124478	days	/1000 / 60 / 60 / 24	17879	5

Please refer to the ***Document Management Strategies*** guide for a more in-depth discussion of this topic.

RESOURCES

Blog: Introducing Couchbase Time Series

Blog: Announcing Vector Search and a Whole Lot More!

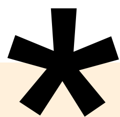
Blog: Data Migration From Oracle to Couchbase

Blog: How to Model Data: A Guide to JSON Data Modeling

Video: Couchbase Data Access and Modeling in the Real World

Video: JSON Data Modeling in Action

Webcast recordings: JSON Data Modeling in Document Databases (part 1), Real-World JSON Data Modeling (part 2)



Modern customer experiences need a flexible database platform that can power applications spanning from cloud to edge and everything in between. Couchbase's mission is to simplify how developers and architects develop, deploy and run modern applications wherever they are. We have reimagined the database with our fast, flexible and affordable cloud database platform Capella, allowing organizations to quickly build applications that deliver premium experiences to their customers – all with best-in-class price performance. More than 30% of the Fortune 100 trust Couchbase to power their modern applications. For more information, visit www.couchbase.com and follow us on X (formerly Twitter) @couchbase.

© 2024 Couchbase. All rights reserved.