Data Modeling Guide

Couchbase
NoEQUAL

Notice and Disclaimer

Data Modeling Guide

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

Couchbase Server is a document database. Unlike traditional relational databases, you store information in documents rather than table rows. Couchbase has a much more flexible data format. 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 that you define. There is no schema in Couchbase; every JSON document can have its own individual set of keys, although you may probably adopt one or more informal schemas for your data.

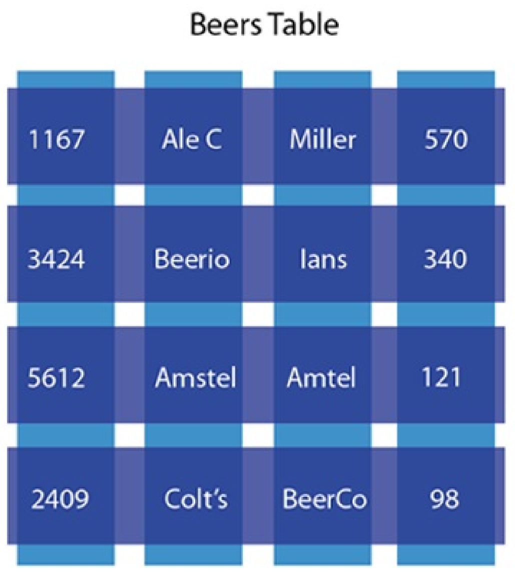
With Couchbase Server, one of the benefits of using JSON documents is that you can index and query these records. This enables you to collect and retrieve information based on rules you specify about given fields; it also enables you to retrieve records without using the key for the record.

Sections

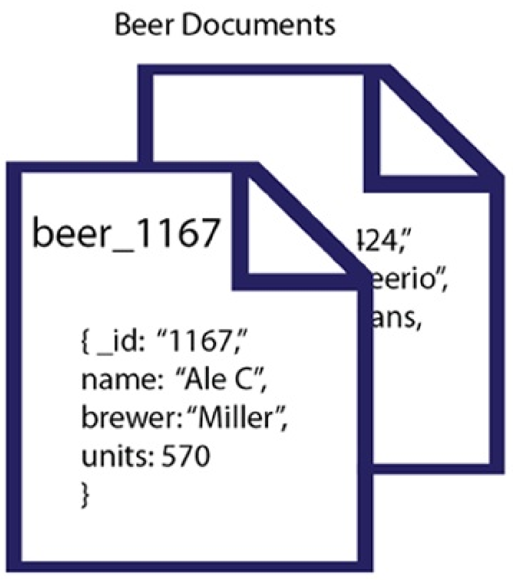
Comparing Document-Oriented and Relational data

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 sub-categories of information about your object. You can also use one or more document 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	Ians	340
5612	Amstel	Amtel	121
2409	Colt's	BeerCo	98


Beer Documents

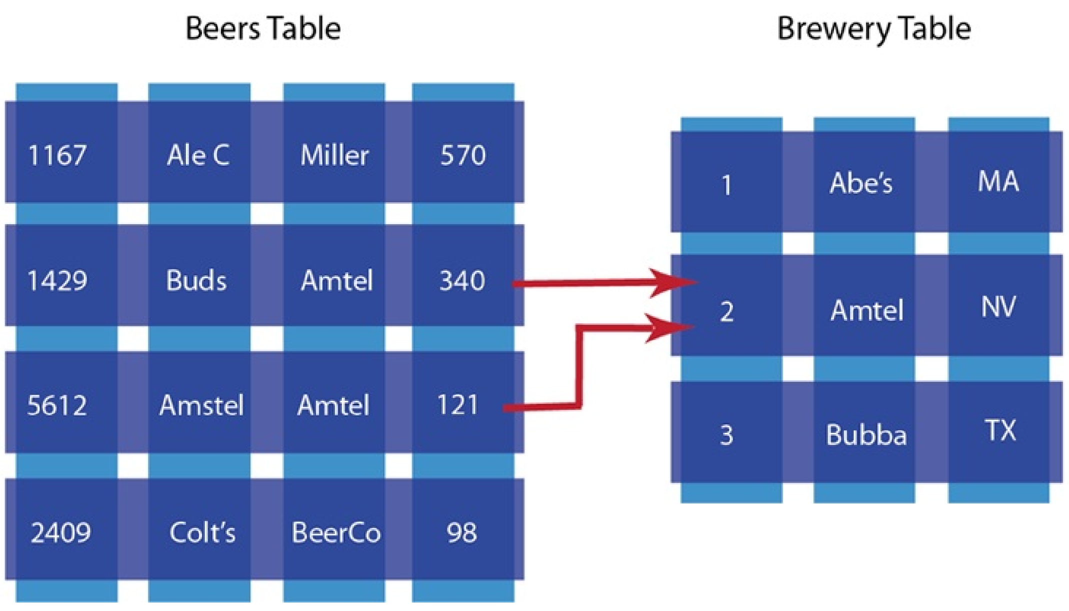
beer_1167

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

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 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 add additional key-value pairs into our documents to represent the new fields.

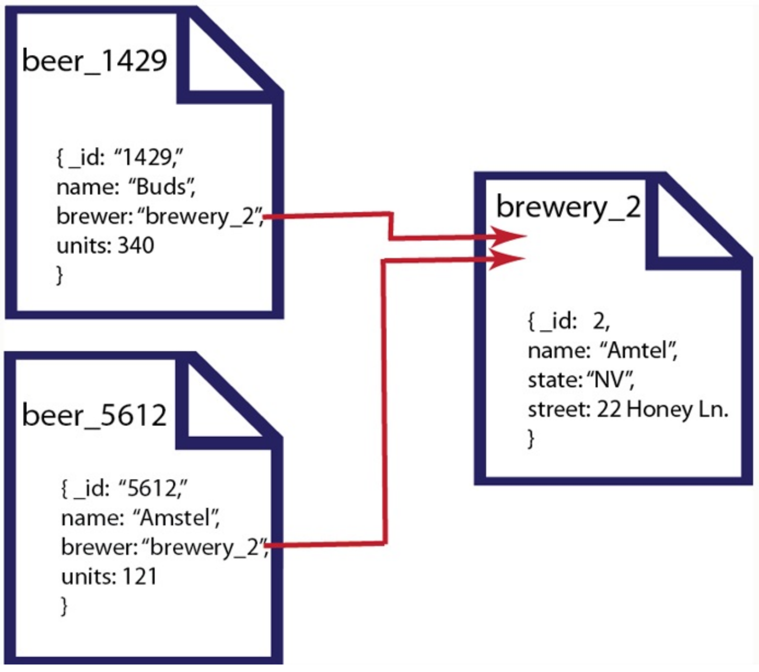
The other characteristic of relational database is *data normalization*; this means you decompose data into smaller, related tables. The figure below illustrates this:



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.

The 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 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. Secondly, we can distribute the information across multiple servers with greater ease. Since records are 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 uses similar constructs to JavaScript. JSON documents enable you to benefit from all the Couchbase features, such as indexing and querying; they also to 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: `["one", "two", "three"]`
- Objects, consisting of key-value pairs, and also known as an *associative array* or hash. The key 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 http://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. Doing data modeling for Couchbase Server is a similar process that you would do for traditional relational databases; there is however much more flexibility and you can change your mind later on your data structures. As a best practice, 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 keys 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 or your application data, 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 pairs 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 Server, 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 key-values pairs, or even nested key-value pairs.

Schemaless Data Modeling

When you use documents to represent data, a database schema is optional. 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 Server 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 Server has its own implicit schema; the schema is represented in how you organize and nest information in your documents.

While you can choose any structure for your documents, the JSON model in particular will help you organize your information in a standard way, and enable you to take advantage of Couchbase Server's ability to index and query. As a developer you benefit in several ways from this approach:

- Extend the schema at run time, 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 is very simple; there are many options, and there are many libraries widely available for JSON conversion.
- Minimization of one-to-many relationships through use of nested entities and therefore, reduction of joins.

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

- Whether you want to use a type field at the highest level of your JSON document in order to group and filter object types.
- What particular keys, ids, prefixes or conventions you want to use for items, for instance **beer_My_Brew**.
- When you want a document to expire, if at all, and what expiration would be best.
- If want to use a document to access other documents. In other words, you can store keys that refer 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*.

You can use a *type* field to group together sets of records. For example, the following JSON document contains a *type* field with the value *beer* to indicate that the document represents a beer. A document that represents another kind of beverage would use a different value in the type field, such as *ale* or *cider*.

```
{
  "beer_id": "beer_Hoptimus_Prime",
  "type": "beer",
  "abv": 10,
  "category": "North American Ale",
  "name": "Hoptimus Prime",
  "style": "Double India Pale Ale"
}
```

Here is another *type* of document in our application which we use to represent breweries. As in the case of beers, we have a type field we can use now or later to group and categorize our beer producers:

```
{
  "brewery_id": "brewery_Legacy_Brewing_Co",
  "type": "brewery",
  "name": "Legacy Brewing Co.",
  "address": "525 Canal Street, Reading, Pennsylvania, 19601 United States",
  "updated": "2010-07-22 20:00:20"
}
```

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 case we decide later that we want to include GPS location of the brewery:

```
{
  "brewery_id": "brewery_Legacy_Brewing_Co",
  "type": "brewery",
  "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
}
```

So in the case of document-based data, we extend the record by just adding the two new fields for *latitude* and *longitude*. When we add other breweries after this one, we would include these two new fields. For older breweries we can update them with the new fields or provide programming logic that shows a default for older breweries. The best approach for 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 Server will send you a message that the data has already changed if someone has already changed the record. For more information about compare and swap methods with Couchbase, see Compare and Swap (CAS).

To create relationships between items, we again use fields. In this example we create a logical connection between beers and breweries using the *brewery* field in our beer document which relates to the *ID* field in 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:

```
{
  "beer_id": "beer_Hoptimus_Prime",
  "type": "beer",
  "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 which brews Hoptimus Prime:

```
{
  "brewery_id": "brewery_Legacy_Brewing_Co",
  "type": "brewery",
  "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 our 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 create relationships between application objects.

Phases of Data Modeling

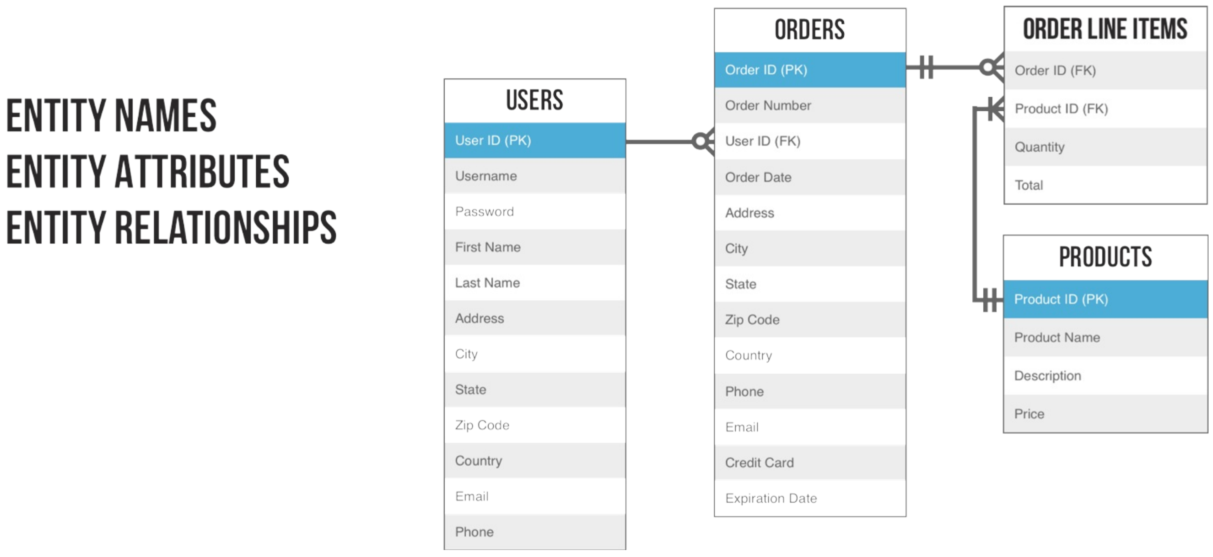
A data modeling exercise typically consists of two phases: **logical data modeling** and **physical data modeling**. Logical data modeling focuses on describing your entities and relationships. Physical data modeling takes the logical data model and maps the entities and relationships to physical containers.

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".

Lets look at some of the key definitions you need from your logical data modeling exercise:

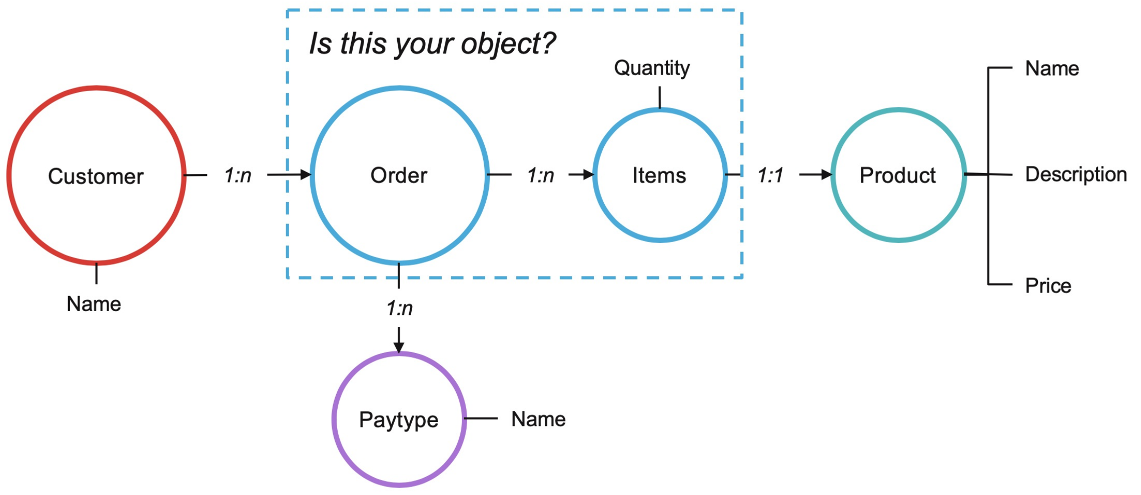


- **Entity keys**: Each entity instance is identified by a unique key. The unique key can be a composite of multiple attributes or a surrogate key generated using a counter or a UUID generator. Composite or compound keys can be utilized to represent immutable properties and efficient processing without 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 attributes *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.

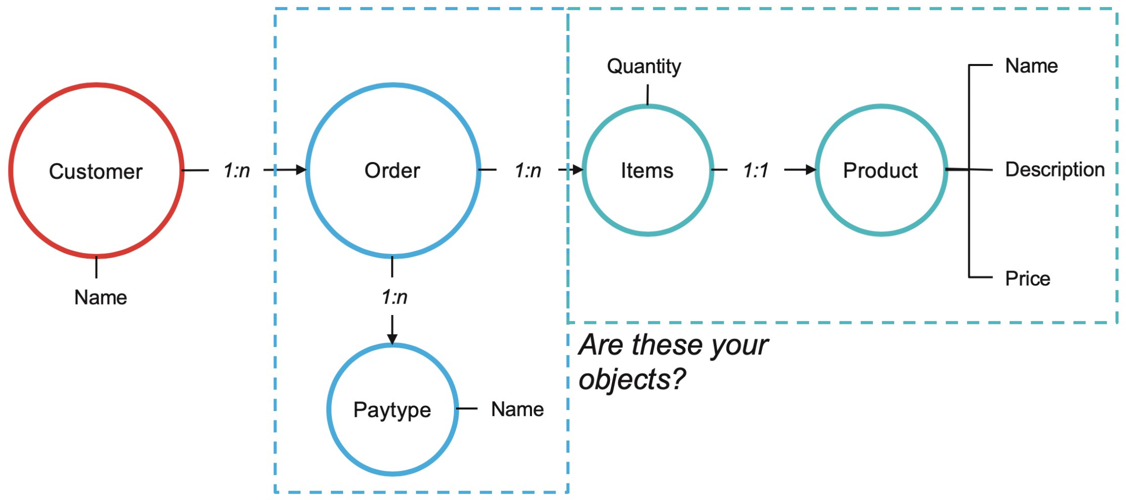
Analyze your *logical* model

Lets look at a highly simplified Order Management System as an example.

In the below diagram: Order embeds Items, and refs external Product (1:n) and Paytype (1:1) docs.



In the below diagram: Order embeds Paytype and refs Items which embeds Product.



Logical data modeling starts with a decision on how to map your entities to documents. JSON documents provide great flexibility in mapping 1-to-1, 1-to-many or many-to-many relationships.

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 & When to Refer

Lets 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:
    - either 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:

```
{
  "orderID": 200,
  "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": "Memorysticks 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. Despite having separate customer profile and item details documents, we replicate 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, though, 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 impacts, 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.
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 despatch team reads the document and updates the status to *Despatched* 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, here 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 finessing 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 despatch address. By embedding the despatch address, as we are, we can easily offer the option to choose a different despatch 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:

```
{
  "orderID": 200,
  "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 three more reads: his user profile (keyed by the email address) and the stock item details (keyed by their item codes).

It requires three 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 data set 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:

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 20 MB upper limit for an individual document but processing the document on the application side would 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 Server, items are used to store associated values that can be accessed with a unique key. Couchbase Server 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 Server is drastically different from relational databases. The following table provides a high level comparison to help you get familiar with Couchbase Server containers.

Data representation and containment in Couchbase Server versus relational databases:

Couchbase Server	Relational databases
Buckets	Databases
Buckets or Items (with type designator attribute)	Tables
Items (key-value or document)	Rows
Index	Index

Items

Items consist of a key and a value. A key is a unique identifier within the bucket. Value can be a binary or a JSON document. You can mix binary and JSON values inside a bucket.

- **Keys**: Each value (binary or JSON) is identified by a unique key. The key is typically a surrogate key generated using a counter or a UUID generator. Keys are immutable. Thus, if you use composite or compound keys, ensure that you use attributes that don't change over time.
- **Values**
  - **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 Server as binary values have a proprietary representation.
  - **JSON values**: JSON provides rich representation for entities. Couchbase Server can parse, index and query JSON values. JSON provide a name and a value for each attribute. You can find the JSON definition at RFC 7159 or at ECMA 404.

The JSON document attributes 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":number,
    "a2":"string",
    "a3":{
       "b1":[ number, number, number ]
    },
    "a4":[
       { "c1":"string", "c2":number },
       { "c1":"string", "c2":number }
    ]
 }
```

JSON Design Choices

Couchbase Server neither enforces nor validates for any particular document structure. 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. Multiple data sets are expected to share a common bucket in Couchbase. To ensure each data set has an isolated keyspace, it is a best practice to include a type/class/use-case/sub-domain prefix in all document keys. As an example of a User Model, you might have a property called `"userId": 123`, the document key might look like `user:123`, `user_123`, or `user::123`. Every Document ID is a combination of two or more parts/values, that should be delimited by a character such as a colon or an underscore. Pick a delimiter, and be consistent throughout your enterprise.

Just as each Document ID should contain a prefix of the type/model, it is also a best practice to include that same value in the body of the document. This allows for efficient filtering by document type at query time or filtered XDCR replications. This property can be named many different names: `type`, `docType`, `_type`, and `_class` are all common choices, choose one that fits your organization's standards.

```
{
  "_type": "user",
  "userId": 123
```

Document Management Fields

At a minimum, every JSON document should contain a type and version property. Depending on your application requirements, use case, the line of business, etc. other common properties to consider at:

- `_created` - A timestamp of when the document was created in epoch time (milliseconds or seconds if millisecond precision is not required)
- `_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)
- `_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

The use of a leading `_` creates a standardized approach to global attributes across all documents within the enterprise.

```
{
  "_type": "user",
  "_schema": "1.2",
  "_created": 1544734688923
  "userId": 123
}
```

The same can be applied through a top-level property i.e. `"meta": {}`.

```
{
  "meta": {
    "type": "user",
    "schema": "1.2",
    "created": 1544734688923
  },
  "userId": 123
}
```

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

Note: There is not a right or wrong property name, however, if you're application will leverage Couchbase Mobile (in particular Sync-Gateway), the use of a leading underscore should be avoided, as any document that contains root level properties with a leading underscore will fail to replicate. This is not a bug, and it meant to facilitate backward compatibility with v1.0 of the replication protocol.

Field name length, style, consistency

- Brevity is beautiful at scale (e.g., 11 vs 6 characters * 1B documents) `geoCode vs countryCode`

- Self-documenting names reduce doc effort/maintenance `userName vs usyslogintxt`
- Consistent patterns reduce bugs (pick and stick to a standard) `firstName or first_name or firstname` , but pick one.
- Use plural names for array fields, and singular for others `"phones": [ ... ], "address": { ... }, "genre": " ... ", "scale": 2.3` .
- Avoid words that are reserved in N1QL (else, escape in N1QL) `user, bucket, cluster, role, select, insert` etc., Please refer N1QL Reserved Word for more details on how to escape reserved words in N1QL.
- Use letters, numbers, or underscore (else, escape in N1QL) `first_name vs first-name` .

Root Attributes vs. Embedded Attributes

The query model changes based on the choice of having a single root attribute or the `type` attribute embedded. Lets take a look at the `track` document as an example.

Root Attributes

Root attribute is a single, top-level attribute with all other attributes encapsulated as an object value of the root attribute. In the below example, the root element of the JSON document is `track` .

```
{
  "track": {
    "artist": "Paul Lekakis",
    "created": "2015-08-18T19:57:07",
    "genre": "Hi-NRG",
    "id": "3305311F4A0FAAFEABD001D324906748B18FB24A",
    "mp3": "https://goo.gl/KgKoR7",
    "ver": "1.0",
    "ratings": [
      {
        "created": "2015-08-20T12:24:44",
        "rating": 4,
        "username": "sublimatingraga37014"
      },
      {
        "created": "2015-08-21T09:23:57",
        "rating": 4,
        "username": "untillableshowings34122"
      }
    ],
    "title": "My House",
    "modified": "2015-08-18T19:57:07"
  }
}
```

Embedded Attributes

In this example, the JSON document is in a flat structure but there is an attribute called `type` embedded within the document.

```
{
  "artist": "Paul Lekakis",
  "created": "2015-08-18T19:57:07",
  "genre": "Hi-NRG",
  "id": "3305311F4A0FAAFEABD001D324906748B18FB24A",
  "mp3": "https://goo.gl/KgKoR7",
  "ver": "1.0",
  "ratings": [
    {
      "created": "2015-08-20T12:24:44",
      "rating": 4,
      "username": "sublimatingraga37014"
    },
    {
      "created": "2015-08-21T09:23:57",
      "rating": 4,
      "username": "untillableshowings34122"
    }
  ],
  "title": "My House",
  "modified": "2015-08-18T19:57:07",
  "_type": "track"
}
```

This is the recommended approach since we can use the `type` field to create index.

```
CREATE INDEX cb2_type ON couchmusic2(_type);

SELECT COUNT(*) AS count
FROM couchmusic2
WHERE _type = "track"
GROUP BY genre;
```

Objects vs. Object Arrays

There are two different ways to represent objects.

- **Objects** - In this choice, `phones` is an object in the `userProfile` class.

```
{
  "type": "userProfile",
  "created": "2015-01-28T13:50:56",
  "dateOfBirth": "1986-06-09",
  "email": "andy.bowman@games.com",
  "firstName": "Andy",
  "gender": "male",
  "lastName": "Bowman",
  "phones": {
    "number": "212-771-1834",
    "type": "cell"
  },
  "pwd": "636f6c6f7261646f",
  "status": "active",
  "title": "Mr",
  "updated": "2015-08-25T10:29:16",
  "username": "copilotmarks61569"
}
```

- **Object Arrays** - In this choice, `phones` is an array of objects in the `userProfile` class.

```
{
  "type": "userProfile",
  "created": "2015-01-28T13:50:56",
  "dateOfBirth": "1986-06-09",
  "email": "andy.bowman@games.com",
  "firstName": "Andy",
  "gender": "male",
  "lastName": "Bowman",
  "phones": [
    {
      "number": "212-771-1834",
      "type": "cell"
    }
  ],
  "pwd": "636f6c6f7261646f",
  "status": "active",
  "title": "Mr",
  "updated": "2015-08-25T10:29:16",
  "username": "copilotmarks61569"
}
```

Array element complexity and use

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

- Store key to lookup/join
  - In this choice, *tracks* is an array of strings which contain track ID's. Let's say we have to get the *track* and *artist* name for each of the track id, in which case we will end up doing multiple *gets*. So,

this choice will have a significant impact when the user base is high, say we have 1M users accessing this information which translates to 3M *gets* for this playlist.

```
{
  "created": "2014-12-04T03:36:18",
  "id": "003c6f65-641a-4c9a-8e5e-41c947086cae",
  "name": "Eclectic Summer Mix",
  "owner": "copilotmarks61569",
  "type": "playlist",
  "tracks": [
    "9FFAF88C1C3550245A19CE3BD91D3DC0BE616778",
    "3305311F4A0FAAFEABD001D324906748B18FB24A",
    "0EB4939F29669774A19B276E60F0E7B47E7EAF58"
  ],
  "updated": "2015-09-11T10:39:40"
}
```

- Or, nest a summary to avoid a lookup/join
  - There are lot of advantages in this approach over the first one. In this choice, 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",
  "id": "003c6f65-641a-4c9a-8e5e-41c947086cae",
  "name": "Eclectic Summer Mix",
  "owner": "copilotmarks61569",
  "type": "playlist",
  "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"
}
```

Timestamp Format

Working with Timestamp format is the difficult thing when it comes to JSON, since JSON does not have a standardized date format. Dates are commonly stored as string in JSON.

The following are examples of commonly used date formats.

- **ISO8601**

```
{
  "countryCode": "US",
  "type": "country",
  "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 extremely useful when you trying to group data. Lets say, you want to generate time series graph and this choice best suits when you want to visualize data.

```
{
  "countryCode": "US",
  "type": "country",
  "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. This is the preferred format when you have to do some kind of date comparison, sorting etc.

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

Four states of data presence in JSON docs

It is important to understand that JSON supports optional properties. If a property has a null value, consider dropping it from the JSON unless there's a good reason not to. N1QL makes it easy to test for missing or null property values. Be sure your application code handles the case where a property value is missing.

- Fields may have a value

  ```
  SELECT geocode WHERE geocode IS VALUED
  ```

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

- Fields may have no value

  ```
  SELECT geocode WHERE geocode IS NOT VALUED
  ```

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

- Fields may be missing

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

  ```
  {
  }
  ```

- Fields may be explicitly null

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

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

Key Design

The most important part of NoSQL database modeling is how do we design our document keys. There are different patterns as mentioned below when it comes to designing a key.

- Prefixing
- Predictable
- Counter ID
- Unpredictable
- Combinations

Prefixing

Multiple data sets are expected to share a common bucket in Couchbase. To ensure each data set has an isolated keyspace, it is a best practice to include a type/class/use-case/sub-domain prefix in all document keys. As an example of a User Model, you might have a property called `"userId": 123`, the document key might look like `user:123`, `user_123`, or `user::123`. Every Document ID is a combination of two or more parts/values, that should be delimited by a character such as a colon or an underscore. Pick a delimiter, and be consistent throughout your enterprise.

- 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`

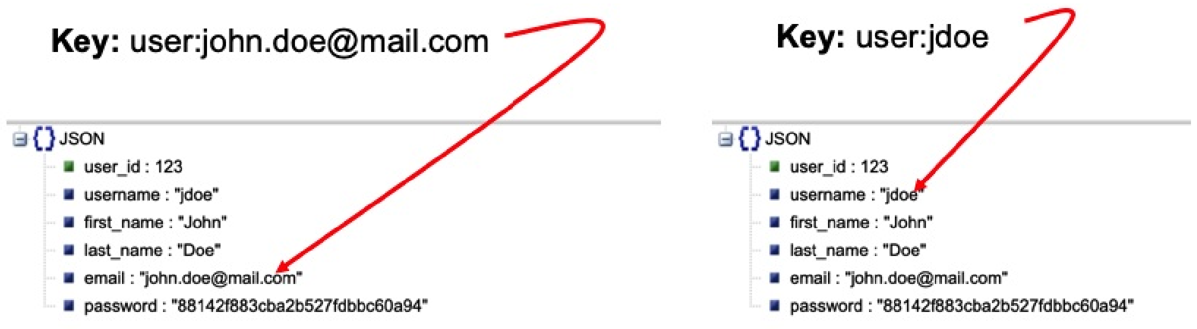
Predictable

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? Well, one thing would be their login name.

So, if we want to make life easy for ourselves in retrieving our user profile, then we can key it with that user's login name. Everything else we need to know about that person could be derived from their user profile, in one way or another.

Pretty quickly we might encounter a problem: for a user to change their login name, we now have to either create a new user profile under a new key or create a look-up document. We could insist that our users can never change their login names 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 : "88142f883cba2b527fdbbc60a94"
```

Key: user:jdoe

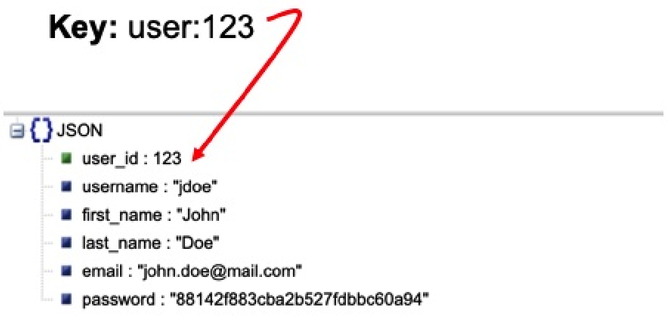
```
{} JSON
   user_id : 123
   username : "jdoe"
   first_name : "John"
   last_name : "Doe"
   email : "john.doe@mail.com"
   password : "88142f883cba2b527fdbbc60a94"
```

Counter ID

We can get Couchbase to generate the key for us using a counter. if you're using a counter ID pattern, every insert (not update) requires 2 mutations. One to increment the counter and the other to mutate the document.

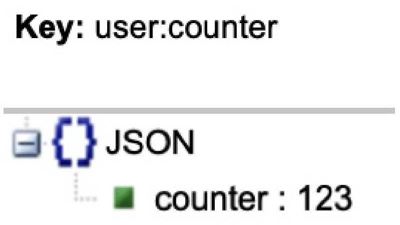
Here's how it works:

1. Someone fills out the new user account form and clicks "Submit".
2. We increment our counter document and it returns the next number up (e.g. 123).
3. We create a new user profile document keyed with 123.
4. We then create 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 : "88142f883cba2b527fdbbc60a94"
```



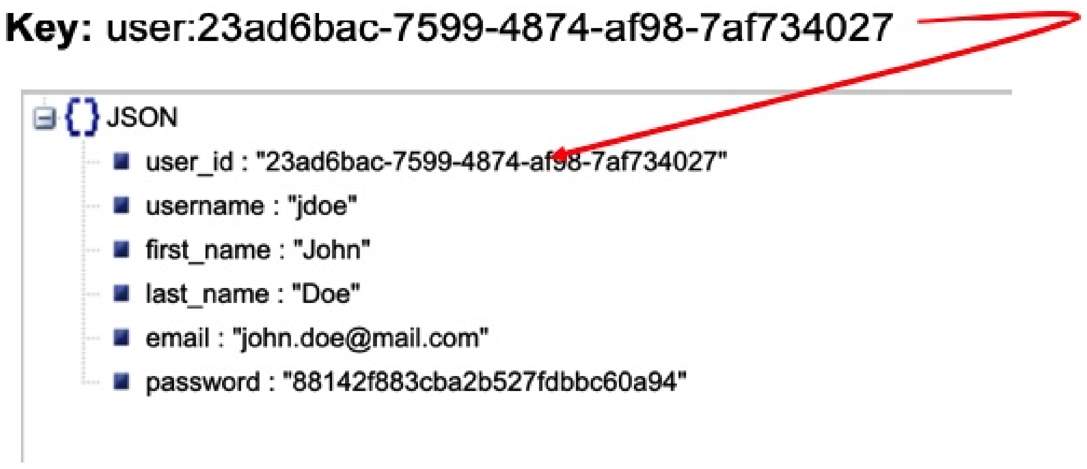
Key: user:counter

```
{} JSON
  counter : 123
```

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

Unpredictable

This pattern uses system generated unique ID's like UUID.



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 : "88142f883cba2b527fdbbc60a94"
```

Combinations

It's when we combine both these methods that we can start to do really interesting things with key names.

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 again. The main document is stored under the key 1001. We're working on an ecommerce site so we also want to know all of the orders our customer has made. Simple: we store the list of orders under the key `1001: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 afresh each time, we instead do it the NoSQL way: we calculate it once and then store it for later retrieval under the key `1001:orders:value` .

Standardized Fields

docType

We have discussed about `docType` in the earlier section above. Please refer *Document Key Prefixing* section in this document for details on docType.

Delimiter

Every Document ID can be a combination of two or more parts/values, that should be delimited by a character such as a colon or an underscore. Pick a delimiter, and be consistent throughout your enterprise.

It is a best practice to only use a single-byte delimiter since this can make a significant difference based on the volume of data.

Schema

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, 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.

```
{
  "_type": "user",
  "_schema": "1.2",
  "userId": 123
}
```

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

Namespacing

The use of a leading `_` creates a standardized approach to global attributes across all documents within the enterprise.

```
{
  "_type": "user",
  "_schema": "1.2",
  "_created": 1544734688923
  "userId": 123
}
```

The same can be applied through a top-level property i.e. `"meta": {}` .

```
{
  "meta": {
    "type": "user",
    "schema": "1.2",
    "created": 1544734688923
  },
  "userId": 123
}
```

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

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, historically this has been abstracted from developers as the schema and the database are managed by a DBA. With an application enforced schema, the model size is now controlled by the application. As developers we tend to be overly verbose when describing variables throughout our applications, 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.

As a general approach, consider the following options to proactively reduce document sizes:

- Don't store the document ID as a repeated value in the document.

- Convert ISO-8601 timestamps to epoch time in milliseconds, saving at least 11 bytes. When millisecond precision is not required, convert to a smaller value (i.e. divide by 1000 to convert to seconds, 60 for minutes, 60 for hours, 24 for days), saving at least 4 bytes.
- Store dates as an ISO format `YYYY-MM-DD` instead of `MMM DD, YYYY` .
- When using GUID's strip all dashes saving an additional 4 bytes per GUID.
- Use shorter property names.
- Don't store properties whose value is `null` , 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.

Storing 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. Generally, this date is stored in 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, this 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 2 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?". Often times 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 *Document Management Strategies* guide for a more in-depth discussion of this topic.

Resources

- JSON Data Modeling for RDBMS Users
- Data Modeling for Couchbase with erwin DM NoSQL
- SQL to JSON Data Modeling with Hackolade
- Moving from SQL Server to Couchbase - Data Modeling