Document Management Strategies

Couchbase
NoEQUAL

Notice and Disclaimer

The recommendations, best practice guides, tuning examples (together "Best Practices") as well as sample code, scripts (together "Sample Code", collectively with the Best Practices is "Content") contained herein are the property of Couchbase, Inc. ("Couchbase") and are provided for illustrative and instructional purposes only. The user of the Content acknowledges and accepts that the Content is not supported by any license agreement between Couchbase and the user.

The Content may not be reproduced, disseminated, sold, sub-licensed, assigned, rented leased, distributed or otherwise published, in whole or in part without prior written permission from Couchbase.

The user of the Source Code assumes the entire risk of any use it may make or permit to be made of the Source Code and is solely responsible for adequate protection and backup of its data. Couchbase reserves the right to make changes to the Source Code or Best Practices at any time without prior notice. ALWAYS thoroughly evaluate Sample Code using test data to ensure proper operation and confirm the Sample Code causes no adverse effects prior to use on live or production data.

Couchbase hereby reserves all rights in the Content under the copyright laws of the United States and applicable international laws, treaties, and conventions.

THE CONTENT HEREIN IS PROVIDED "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. WITHOUT LIMITING ANY OF THE FOREGOING AND TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT SHALL Couchbase OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) SUSTAINED BY YOU OR A THIRD PARTY, HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT ARISING IN ANY WAY OUT OF THE USE OF THE CONTENT, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. The foregoing shall not exclude or limit any liability that may not by applicable law be excluded or limited.

Use of or access to Couchbase products or services requires a separate license from Couchbase.

Document Management Strategies

JSON provides a flexible data model, which can support an infinite number of schemas, as the schema is explicitly stored alongside every value. Every application evolves over time, schemas change, new models are defined. It is important to have a plan for managing and adapting these changes into your data model and applications smoothly, this document is intended to explain best practices and conceptual implementations of how this might work.

> **Note:** All code examples included are in pseudo-code. They are provided to relay the logical concept only. The conceptual high-level logic and JSON properties should be adapted to your organizational coding standards and programming language.

Standardized Document Properties

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
}
```

> **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.

Applications are typically versioned using Semantic Versioning, i.e. 2.5.1. Where 2 is the major version, 5 is the minor version and 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:

> "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 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
}
```

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.

Document 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.

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

Embedded vs. Non-Embedded Data

Typically denormalized document models provide better read performance

Embed when there are:

- Relationship between entities
- One-to-few relationships between entities
- Embedded data that will not grow unbounded
- Embedded data that is integral to data in a document

Do not embed and normalize when there are:

- Unbounded data/arrays
- Frequent change of data across models
- Unrelated models
- One-to-many relationships
- Many-to-many relationships
- Frequent changes to related data
- Referenced data could be unbounded
- Smaller mutations are required for replication/network performance

---

Schema Versioning

Storing the schema version within the document is a best practice that provides a mechanism to migrate and upgrade models as they change over time. For this example, the pseudo-code provided is intended to illustrate an approach to schema management through code. There are many different ways to solve this problem, think of this exercise as things to consider during development, not necessarily how it is coded.

Document ID: user:123

Document Body:

```
{
  "_type":"user",
  "_schema": "1.0",
  "_created": 1544759124,
  "userId": 123,
  "name": "Joe Smith",
  "phone": "1234567890",
  "email": "joe.smith@acme.com"
```

```
}
```

The easiest approach to begin managing documents is to use a Class to represent a Document, this should be a 1:1 relationship. Funneling all operations at the document level to a single class enables you to make rapid changes, whereas if changes to the document are allowed throughout the codebase any schema changes will subsequently require more code modifications and testing.

```
public class User {
  var _type: string
  var _schema: numeric
  var _created: datetime
  var _modified: datetime
  var userId: integer
  var name: string
  var phone: string
  var email: string
}
```

Next, we need to add the constructor

```
public class User {
  var _type: string
  var _schema: numeric
  var _created: datetime
  var _modified: datetime
  var userId: integer
  var name: string
  var phone: string
  var email: string

  constructor (doc) {
    this._type = doc._type
    this._schema = doc._schema
    this._created = doc._created
    this._modified = doc._modified
    this.userId = doc.userId
    this.name = doc.name
    this.phone = doc.phone
    this.email = doc.email
  }
}
```

Now that the user class is defined, it can start to be used and it's instance properties referenced directly, for example:

```
user = new User(...) // create a new instance of User
print(user.phone) // output the users phone
user.phone = "111-222-3333" // update the users phone number
```

To control our schema, and how it is consumed throughout the application, direct references should not be allowed, instead, the values exposed from the document are better served through accessors (i.e. getters and setters). The use of accessors is highly beneficial, as a method is used to access the value or modify the value. The method can apply business rules and can be changed without impacting existing consumers.

```
public class User {
  private var _type: string
  private var _schema: numeric
  private var _created: datetime
  private var _modified: datetime
  private var userId: integer
  private var name: string
  private var phone: string
  private var email: string

  constructor (doc) {
    this.set_Type(doc._type)
    this.set_Schema(doc._schema)
    this.set_Created(doc._created)
    this.set_Modified(doc._modified)
    this.setUserId(doc.userId)
    this.setName(doc.name)
    this.setPhone(doc.phone)
    this.setEmail(doc.email)
  }
  ...
  public getPhone() {
    // return the phone formatted as: (111) 111-1111
    return "(" + this.phone.substring(0, 3) + ") " +
      this.phone.substring(3, 6) + "-" +
      this.phone.substring(6)
  }
  public setPhone(value string) {
    value = value.replace("[^:digit:]", "") // remove any non-numeric charac
ters
    // validate the phone
    if (value.length != 10) {
      throw("invalid phone number")
    }
    this.phone = value
  }
```

```
  ...
}
```

Using the accessors, instead of references would look similar to:

```
user = new User(...) // create a new instance of User
print(user.getPhone()) // output the users name
user.setPhone("111-222-3333") // update the users phone number
```

As this functionality will be common across all of the documents within our domain, centralize these shared properties and methods into a Base class that all document classes extend.

```
public class Base {
  private var _type: string
  private var _schema: numeric
  private var _created: datetime
  private var _modified: datetime

  constructor (doc) {
    this.populate(doc) // load the document
  }
  private populate(doc) {
    // loop doc properties and dynamically call set accessors
    // loading the entire document
  }
  ...
  public get_Modified() {
    return this._modified;
  }
  public set_Modified(value datetime) {
    this._modified = value
  }
}

public class User extends Base {
  private var userId: integer
  private var name: string
  private var phone: string
  private var email: string

  constructor User(doc) {
    super(doc)
    this.set_Type("user")
    this.set_Schema("1.0")
  }
```

```
  ...
  public getName() {
    return this.name;
  }
  public setName(value string) {
    this.name = value
  }
}
```

Evolving the Schema

Let's say at some point in the near future, the decision is made to split the `name` field out into first and last name. This is a simple update to the document structure, as follows:

```
{
  "_type":"user",
  "_schema": "2.0",
  "_created": 1544759124,
  "userId": 123,
  "firstName": "Joe",
  "lastName": "Smith",
  "phone": "1234567890",
  "email": "joe.smith@acme.com"
}
```

Easy enough, but now we have to deal with the structure change in the data objects:

```
public class User extends Base {
  private var userId: integer
  private var firstName: string
  private var lastName: string
  private var phone: string
  private var email: string

  constructor User(doc) {
    super(doc)
    this.set_Type("user")
    this.set_Schema("2.0")
  }
}
```

Simple right? Only, what happens when you try to load a user document that's still formatted for version 1.0? In this case, we will introduce a `migrate()` into our class that is the first method called from the constructor

```
public class User extends Base {
  private var userId: integer
  private var firstName: string
  private var lastName: string
  private var phone: string
  private var email: string

  constructor User(doc) {
    doc = migrate(doc) // transform the document
    super(doc)
    this.set_Type("user")
    this.set_Schema("2.0")
  }
  private migrate(doc){
    if(doc._schema == "1.0") {
      this.setFirstName(doc.name.split(" ")[0])
      this.setLastName(doc.name.split(" ")[1])
      delete doc.name
      doc._schema = "2.0"
    }
  }
}
```

Our `migrate()` method, now has the business rules in place to transform a version 1.0 document to a version 2.0 document. As part of our code changes, we would've introduced accessors for the new properties: `getFirstName()`, `setFirstName()`, `getLastName()`, `setLastName()`. But what about the old accessors? Should they be removed? They could be removed, but then all code referencing those methods would need to be refactored as well, it is safer to just update their implementation:

```
  public getName() {
    return this.getFirstName() + " " + this.getLastName()
  }

  public setName(name) {
    setFirstName(name.split(" ")[0])
    setLastName(name.split(" ")[1])
  }
```

By maintaining the older accessor methods, you're protecting against any code locations that are missed during the refactoring and are still calling the older methods.

Further Evolutions

Now let's say that the phone property is changed from a single value to a list of multiple phone numbers:

```
{
  "_type":"user",
  "_schema": "3.0",
  "_created": 1544759124,
  "userId": 123,
  "firstName": "Joe",
  "lastName": "Smith",
  "phones": [
    {
      "type": "mobile",
      "number": "1234567890"
    }
  ]
}
```

Now you've got to extend the modifications routines for this additional schema change:

```
public class User extends Base {
  private var userId: integer
  private var firstName: string
  private var lastName: string
  private var phones: array
  private var email: string

  constructor User(doc) {
    doc = migrate(doc) // transform the document
    super(doc)
    this.set_Type("user")
    this.set_Schema("3.0")
  }
  ...
  public getPhones() {
    return this.phones
  }
  public setPhones(value) {
    return this.phones = value
  }
  public addPhone(type, phone) {
    // validate the type
    if (!["other", "mobile", "home", "work"].contains(type)) {
      throw("invalid type")
    }
    phone = phone.replace("[^:digit:]", "") // remove any non-numeric charac
ters
    // validate the phone
```

```
      if (phone.length != 10) {
        throw("invalid phone number")
      }
      this.phones.push({
        type: type,
        phone: phone
      })
    }
    public getPhone() {
      // return the phone formatted as: (111) 111-1111
      var phone = this.getPhones()[0];
      return "(" + phone.substring(0, 3) + ") " +
        phone.substring(3, 6) + "-" +
        phone.substring(6)
    }
    public setPhone(value string) {
      addPhone("other", value)
    }
    ...
    private migrate(doc){
      if(doc._schema == "1.0") {
        doc = migrateFromV1toV2(doc)
      }
      if(doc._schema == "2.0") {
        doc = migrateFromV2toV3(doc)
      }
    }
    private migrateFromV1toV2(doc){
      this.setFirstName(doc.name.split(" ")[0])
      this.setLastName(doc.name.split(" ")[1])
      delete doc.name
      doc._schema = "2.0"
    }
    private migrateFromV2toV3(doc){
      this.addPhone("other", doc.phone)
      delete doc.phone
      doc._schema = "3.0"
    }
  }
```

We're no longer dealing with a scalar value, but a complex array/list. Based on your access patterns and use case this has the potential to be more involved. For example along with property specific accessors `getPhones()` and `setPhones()` you may want to have `addPhone()`, `removePhone()`, `updatePhone()`, etc.

Summary

So, as you've seen, by adding a schema version number to the document, there are ways of handling in the code the migrations of documents from the older versions to the newer, without having to perform a massive data conversion on your Couchbase bucket. You have a choice of doing an at request time migration, or you could still choose to do a mass migration of data but the rules for that migration are in reusable code. This is primarily concerned with data values that change the data type, are deprecated from the document model, or other changes that your application needs to deal with as your data model evolves.

In summary, you need to:

- Put some metadata into your documents, such as the document type and version.
- In your data object loader/serialization method, include functionality to migrate the document from one version to the next.
- Be sure to keep the schema migration routines in order, as new revisions are created and the schema evolves.

Document Revisions

From time to time, there may be a requirement to save multiple revisions of a document within a bucket, this history is maintained by the application. This section is intended to outline one way this can be accomplished.

Document Metadata

Just as we've shown in previous sections, we'll add a standardized metadata property to our document model to assist with storing the revision information. A certain amount of this is standard practice around the industry. Building on our User model, we can add a `_ver` property to document the current version of the document.

```
{
  "_type":"user",
  "_schema": "3.0",
  "_ver": 1,
  "_created": 1544759124,
  "userId": 123,
  "firstName": "Joe",
  "lastName": "Smith",
  "phones": [
    {
      "type": "mobile",
      "number": "1234567890"
    }
  ]
}
```

Then, in your data object, you would want to add some code to initialize and increment this revision number as updates are made to a document.

```
public class User extends Base {
  private var _ver: integer
  private var userId: integer
  private var firstName: string
  private var lastName: string
  private var phones: array
  private var email: string

  constructor User(doc) {
    doc = migrate(doc) // transform the document
    super(doc)
    this.set_Type("user")
    this.set_Schema("3.0")
    // set _ver if it is not defined
    if (!this._ver) {
      this._ver = 1
    }
  }
  ...
}
```

Maintaining Revisions

Because you are wanting to maintain the revisions of a document, this requires steps to be taken with writing updates to a document to preserve the previous version. It also requires using a predictable pattern in the revision key generation to make it easy to find and retrieve specific document revisions. One of the simplest ways of using a predictable pattern in the key generation is to append the revision number to the document key, so if the document had a key of `user:123`, the revision might have a key like `user:123:v:1`. So, if a revision had been made to our example user document, you'd have two different documents:

Document ID: `user:123`

```
{
  "_type":"user",
  "_schema": "3.0",
  "_ver": 2,
  "_created": 1544759124,
  "userId": 123,
  "firstName": "Joe",
  "lastName": "Smith",
  "phones": [
    {
      "type": "mobile",
```

```
      "number": "1234567890"
    },
    {
      "type": "home",
      "number": "1234445555"
    }
  ]
}
```

and

Document ID: `user:123:v:1`

```
{
  "_type":"user",
  "_schema": "3.0",
  "_ver": 1,
  "_created": 1544759124,
  "userId": 123,
  "firstName": "Joe",
  "lastName": "Smith",
  "phones": [
    {
      "type": "mobile",
      "number": "1234567890"
    }
  ]
}
```

This will require code in your objects `save()` method to make a copy of the current document prior to saving the update:

```
public save() {
  this.copyRevision()
  this._ver++
  bucket.upsert(
    this._type + ":" + this.userId, // key
    this // value
  )
}

private copyRevision() {
  var doc = bucket.get(this._type + ":" + this.userId)
  bucket.insert(
    this._type + this.userId + ":v:" + this._ver, // key
```

```
    doc // value
  )
}
```

Limiting Revisions

The problem with maintaining document revisions is that they can significantly increase the amount of storage space needed to hold them. It's not unusual for a document to be updated thousands of times over its lifespan. So odds are that you'll want to limit the number of revisions to be kept. To avoid having to go back and purge outdated revisions, the better solution would be to determine ahead of time what your revision limit is going to be, and to build that into your data objects:

```
private maxRevisionCount = 10

public class User extends Base {
  ...
}
```

Then you would want to perform a check before saving a revision to see if an older revision needs to be deleted:

```
public save() {
  if (this._ver >= maxRevisionCount) {
    this.deleteOldRevision()
  }
  this.copyRevision()
  this._ver++
  bucket.upsert(
    this._type + ":" + this.userId, // key
    this // value
  )
}

private deleteOldRevision() {
  bucket.delete(this._type + ":" + this.userId + ":v:" + (this._ver - maxRev
isionCount))
}
```

Summary

By adding a revision number and using a key generation scheme that appends the revision number to the current revision key in a predictable way, it can be fairly straightforward to implement a document revision retention policy.

In summary, you'll need to:

- Add a revision number property to the document and data object.
- Increment the revision number property each time the document is updated.
- Copy the prior revision document, adding the revision number to the document key in a predictable way prior to saving any updates to the current revision of the document.
- Hard-code a maximum number of revisions to be kept.
- If the revision number of the document is greater than the maximum number of revisions being kept, subtract the maximum number of revisions from the current revision number and delete that version of the document when saving any updates to the current revision.