NoSQL design (part 3)

Data Management and Visualization

Politecnico di Torino

Ex. 1 – Tracking document versions

In some systems, rather than **updating** an existing object and **overwriting** its various attributes there is a business requirement to preserve the original document and to create a new **version** of this document, instead of updating it.

```
{    docId: "A",
     v: 1,
     color: "red",
     locale: "USA"
}
```

Challenges:

- Correctly generate the new **version number** in a distributed system
- Return only the **current (latest)** version of each document when there is a query
- If the system **fails** at any point, you must either have a consistent state of the data, or it must be possible on re-start to infer the state of the data and clean it up, or otherwise bring it to a consistent state.

http://www.askasya.com/post/trackversions/

Design approaches

- Full **embedding**: store all document versions as a list inside a single document
- Version attribute: store a **full new document** at each update, with monotonically increasing **version number**
  - Variant: add a field in latest version identifying it as such
- **Separate collections**: store current version in “primary” collection and previous versions in a second collection
- **Differential approach**: store only “deltas”, in the same document

Full embedding

Store all document versions as a list inside a single document

```
{   "_id" : 174,
     "current" : { "v" :3, "attr1": 184, "attr2": "A-1" },
     "prev" : [
                  { "v" :1, "attr1": 165 },
                  { "v" :2, "attr1": 165, "attr2": "A-1" }
               ]
}
```

- The current state is represented by the “current” subdocument.
- How many versions you expect to handle and how long you have to keep them? How large is a document?
- Get current version with:
  `db.collection.find({"docId":174}, {"prev":0})`

Full embedding

Store all document versions as a list inside a single document

```
{   "_id" : 174,
     "current" : { "v" :3, "attr1": 184, "attr2": "A-1" },
     "prev" : [
                  { "v" :1, "attr1": 165 },
                  { "v" :2, "attr1": 165, "attr2": "A-1" }
               ]
}
```

Update:

- read the current document,
- move “current” field to the end of the previous array,
- set the new “current” field appropriately and save to db

Full embedding

Pros

- **atomic updates** of a single document (single operation sets new current and updates previous)
- very **simple querying**
  - it only needs to request the “current” attribute,
  - easily access or discard all previous versions
- the (typically builtin) “_id” index can be used for the unique document id **preventing duplicates** being accidentally inserted
- creating the **next version number** is simple and thread-safe

Full embedding

Cons

- when a document grows beyond its previously allocated size, it has to be moved on disk (updates becomes more computationally expensive)
- unsuitable for documents that have many (e.g., thousands) of **versions** over their **lifetime**
  - the documents would get too big, and could potentially exceed the document-size limit , e.g., 16 Megabytes in MongoDB
  - also the update rate (i.e., number of updates/second/document) impacts the suitability of the solutions
  - suboptimal disk access, i.e., read and discard large data on disk (all versions)
- likely more **fragmentated data** with respect to the “naive” approach of inserting new versions as new documents
- if the system has a very **high level of concurrency**, when multiple threads/clients are trying to make different updates to the same document, a single thread could keep failing and it might take **multiple re-tries** to persist its update

New document with new version

Requirement

- Keep track of **any previous version** of a document that ever existed in a particular collection

If we need to set color to “blue”, instead of updating the “color” field from “red” to “blue”, we create a **new document**.

Original doc

```
{    docId: "A",
     v: 1,
     color: "red",
     locale: "USA"
}
```

New doc

```
{    docId: "A",
     v: 2,
     color: "blue",
     locale: "USA"
}
```

New document with new version

Original doc

```
{    docId: "A",
     v: 1,
     color: "red",
     locale: "USA"
}
```

New doc

```
{    docId: "A",
     v: 2,
     color: "blue",
     locale: "USA"
}
```

Create a **unique index** on `{"docId":1,"v":1}`

- avoids duplicate versions
- failure does not create inconsistent states

Query the **latest version** with

```
db.docs.find(
   {"docId":"A"}).sort(
   {"v":-1}).limit(-1);
```

New document with new version + current

Original doc

```
{    docId: "A",
     v: 1,
     color: "red",
     locale: "USA“,
     current: false
}
```

New doc

```
{    docId: "A",
     v: 2,
     color: "blue",
     locale: "USA“,
     current: true
}
```

Query the **latest version** with

```
db.docs.find(
   {"docId":"A“,
    “current”: true})
```

New document with new version + current

Original doc

```
{    docId: "A",
     v: 1,
     color: "red",
     locale: "USA“,
     current: false
}
```

New doc

```
{    docId: "A",
     v: 2,
     color: "blue",
     locale: "USA“,
     current: true
}
```

- Updating becomes difficult now, since there is no method to insert one document and update another document “as one”.
- Typical optimistic approach:
  - first insert a new version, based on currently highest version
  - then update the previously current document to $unset the “current” field
- If our process fails between those two write operations, we will have two documents with {“current”:true}

Separate collections

Store current document in your “primary” collection, and keep older versions in another collection.

CollectionOfCurrentDocuments:

```
{"docId": 174, "v":3, "attr1": 184, "attr2" : "A-1" }
```

CollectionOfPreviousVersions:

```
{"docId": 174, "v": 1, "attr1": 165 }
{"docId": 174, "v": 2, "attr1": 165, "attr2": "A-1" }
```

This one is the simplest of them all. Since the old versions are in another collection, you just query normally when you need to find a single or multiple documents - they will all be the current version.

Separate collections

Store current document in your “primary” collection, and keep older versions in another collection.

CollectionOfCurrentDocuments:

```
{"docId": 174, "v":3, "attr1": 184, "attr2" : "A-1" }
```

CollectionOfPreviousVersions:

```
{"docId": 174, "v": 1, "attr1": 165 }
{"docId": 174, "v": 2, "attr1": 165, "attr2": "A-1" }
```

- Conceptually the **simplest** solution to **get the current** document version
- Since the old versions are in another collection, you just query normally when you need to find a single or multiple documents

Separate collections

Store current document in your “primary” collection, and keep older versions in another collection.

CollectionOfCurrentDocuments:

```
{"docId": 174, "v":3, "attr1": 184, "attr2" : "A-1" }
```

CollectionOfPreviousVersions:

```
{"docId": 174, "v": 1, "attr1": 165 }
{"docId": 174, "v": 2, "attr1": 165, "attr2": "A-1" }
```

- The **hardest** solution to **write a new version**
- Read the “old” current document,
- Apply the updates, hence creating the “new” current document
- Save the new current document on top of the previous one and if successful, then insert the old current document into the “previous” collection.

Separate collections

- The **hardest** solution to **write a new version**
- Read the “old” current document,
- Apply the updates, hence creating the “new” current document
- Save the new current document on top of the previous one and if successful, then insert the old current document into the “previous” collection.

The **problem**

- meanwhile you may fail, e.g., before that last write
- you’ll be missing a version of the document from the “previous” collection

Separate collections

The **problem**

- meanwhile you may fail, e.g., before that last write
- you'll be missing a version of the document from the "previous" collection

A possible **solution**

- Inverse write order: we write the doc to previous collection first, then we save it into "current" collection
- if someone else is trying to update this document, they will also be saving into "previous" collection, so having a unique index on (document id, version) will tell us if we lost the race and try again
- if the thread dies in the middle (after insert into previous and before updating current) it's not the end of the world, as your current collection was not affected, but you can have a fix to "clean up" your "previous" collection

Differential approach

Store only the differences with increasing versions.

Example with “new doc – new version” approach:

```
{"docId": 174, "v": 1, "attr1": 165 }
{"docId": 174, "v": 2, "attr1": 165, "attr2": "A-1" }
{"docId": 174, "v": 3, "attr1": 184, "attr2": "A-1" }
```

Example with differential approach:

```
{"docId": 174, "v": 1,  "attr1": 165 }
{"docId": 174, "v": 2,  "attr2": "A-1" }
{"docId": 174, "v": 3,  "attr1": 184 }
```

Differential approach

Store only the differences with increasing versions.

```
{"docId": 174, "v": 1,  "attr1": 165 }
{"docId": 174, "v": 2,  "attr2": "A-1" }
{"docId": 174, "v": 3,  "attr1": 184 }

{"docId": 174, "v": 3, "attr1": 184, "attr2": "A-1" }
```

The current version is hard to get

- must be derived by “merging” all the documents with the same `docId`, keeping the “latest” or “highest” version’s value of each attribute if it occurs in more than one version

Updates are simpler

- however, the right version number has to be generated for keeping track of the latest value of each attribute

Differential approach

Store only the differences with increasing versions.

```
{"docId": 174, "v": 1,  "attr1": 165 }
{"docId": 174, "v": 2,  "attr2": "A-1" }
{"docId": 174, "v": 3,  "attr1": 184 }

{"docId": 174, "v": 3, "attr1": 184, "attr2": "A-1" }
```

The current version is hard to get

- must be derived by “merging” all the documents with the same `docId`, keeping the “latest” or “highest” version’s value of each attribute if it occurs in more than one version

Updates are simpler

- however, the right version number has to be generated for keeping track of the latest value of each attribute

Exercise (mutable documents)

Trucks in a company fleet are driven by different drivers in different days. They transmit

- location,
- fuel consumption,
- (and other operating metrics)

every 60 seconds to a fleet management database.

Identify the truck of the sensor reading, the time and date, the driver, and the metrics.

Exercise (mutable documents)

```
Example
{
 truck_id: 'T87V12',
 time: '08:10:00',
 date : '27-May-2015',
 driver_id: ‘D123456’,
 driver_name: ‘Jonh Doe’,
 fuel_consumption_rate: '14.8 mpg’,
 location: [lat, long],
 …
}
```