NoSQL design (part 2)

Data Management and Visualization
Politecnico di Torino
Ex. 1 - books and publishers

Design a NoSQL document-based database to store the following data about books.

The number of books is not known in advance and it is set to have a virtually unlimited grow.

The number of publishers is limited.

- Title, e.g., «MongoDB Guide»
  - Each book has only one title
  - Different books might have the same title
- Authors’ surnames
  - A possibly long list, bounded by real-life constraints
- Publication date
- Total number of pages
- Language of the text
- Publisher data
  - name,
  - year of foundation,
  - U.S. state where headquarters are located
Books and publishers (embedding)

```json
{
    _id: 123456789,
    title: "MongoDB Guide",
    authors: [ "Chodorow", "Dirolf" ],
    published_date: ISODate("2010-09-24"),
    pages: 216,
    language: "English",
    publisher: {
        name: "O'Reilly Media",
        founded: 1980,
        location: "CA"
    }
}
```

Let’s suppose that accessing a book, its publisher data must be known.

- The «publisher» information (subdocument) is repeated across all books sharing the same publisher.

Books and publishers (referencing)

Let’s suppose that accessing a publisher, all its books must be known.

- Use references
- The growth of the relationships determine where to store the reference
- If the number of books per publisher is small with limited growth, store the book reference inside the publisher document
Books and publishers (referencing)

Let’s suppose that accessing a publisher, all its books must be known.

- Use references
- The growth of the relationships determine where to store the reference
  - If the number of books per publisher is small with limited growth, store the book reference inside the publisher document
  - if the number of books per publisher is unbounded, this data model would lead to mutable, growing arrays
Books and publishers (referencing)

```json
{
  _id: 123456789,
  title: "MongoDB Guide",
  authors: [ "Chodorow", "Dirolf" ],
  published_date: ISODate("2010-09-24"),
  pages: 216,
  language: "English",
  publisher_id: "oreilly"
}
```

```json
{
  _id: "oreilly",
  name: "O'Reilly Media",
  founded: 1980,
  location: "CA",
  books: [123456789, 234567890, ...]
}
```

- To avoid mutable, growing arrays, store the publisher reference inside the book document
- Each document only has 1 publisher, size of the "publisher_id" attribute cannot grow
- Accessing the publisher and all its books requires two fetches
  - The publisher document with the desired _id
  - A query on the book documents with the desired publisher_id
Ex. 2 - Blog posts and comments

Design a NoSQL document-based database to store the following data about blog posts and their comments.

When accessing a blog post, all its comments must be visualized together.

Comments are added to blog posts by blog readers.

- Post title
- Post author
  - Suppose one author per post
- Post content: text
- Tags of the post: one or more
- Comments to the blog post
  - Author of the comment
  - Comment text
Blog posts and comments (embedding)

```
{
    "_id": "myslug",
    "_rev": "123456",
    "author": "john",
    "title": "My blog post",
    "content": "Bla bla bla …",
    "tags": ["tag1", "tag2", ...],
    "comments": [
        {"author": "jack", "content": "..."},
        {"author": "jane", "content": "..."}
    ]
}
```

• The application can get and visualize the whole blog post with only 1 request to the database
• Deleting the blog post consistently delete also its comments
• To add a comment:
  • get blog-post doc
  • add to comment list
  • save back the doc to db
  • possible conflicting update (if no atomic addToList)
  • unbounded growth!

Blog posts and comments (referencing)

```
{
    "id": "post1",
    "rev": "123456",
    "type": "post",
    "author": "john",
    "title": "My blog post",
    "content": "Bla bla bla ...",
    "tags": ["tag1", "tag2", ...],
}
```

- Separate documents (in the same collection) for blog posts and post comments
- Type attribute
- To get the whole blog post, multiple fetches to the database are required...

```
{
    "id": "ABCDEF",
    "rev": "123456",
    "type": "comment",
    "post": "post1",
    "author": "jack",
    "content": "...
}
```

```
{
    "id": "DEFABC",
    "rev": "123456",
    "type": "comment",
    "post": "post1",
    "author": "jane",
    "content": "...
}
```
Blog posts and comments (referencing)

```json
{
  "_id": "post1",
  "_rev": "123456",
  "type": "post",
  "author": "john",
  "title": "My blog post",
  "content": "Bla bla bla …",
  "tags": ["tag1", "tag2", ...],
}

• Single fetch by means of MapReduce

```javascript
function(doc) {
  if (doc.type == "post") {
    map([doc._id, 0], doc);
  } else if (doc.type == "comment") {
    map([doc.post, 1], doc);
  }
}
```

```json
{
  "_id": "ABCDEF",
  "_rev": "123456",
  "type": "comment",
  "post": "post1",
  "author": "jack",
  "content": "...

}, {
  "_id": "DEFABC",
  "_rev": "123456",
  "type": "comment",
  "post": "post1",
  "author": "jane",
  "content": "...

}
```

Might be `doc.timestamp` to sort
Blog posts and comments (bucketing)

• Even with MapReduce, if we have 1000 comments on a blog post, we would need to retrieve all 1000 documents causing a lot of reads by the database

• Let’s try to balance the rigidity of the embedding strategy with the flexibility of the referencing strategy

• Split the comments into buckets with a maximum of 50 comments in each bucket

• Typical cases where bucketing is appropriate are ones such as bucketing data by hours, days or number of entries on a page (like comments pagination).

{ 
    blog_entry_id: 1, 
    page: 1, 
    count: 50, 
    comments: [ 
        { 
            name: "Peter Critic", 
            comment: "Awesome post" 
        }, ...] 
} 

{ 
    blog_entry_id: 1, 
    page: 2, 
    count: 1, 
    comments: [ 
        { 
            name: "John Page", 
            comment: "another comment" 
        } 
    ] 
}
Ex. 3 - Books and authors

Design a database to store data about books and their authors.

When accessing a book, also information on the authors must be retrieved.

When accessing an author, also information on her/his books must be retrieved.

- Books
  - Title
  - Categories
  - Authors

- Authors
  - Name
  - Books written
Books and authors

- Authors

  ```json
  { 
    _id: 1,
    name: "John Doe",
    books: [1, 2]
  }

  { 
    _id: 2,
    name: "Christopher Wu",
    books: [2]
  }
  ```

- Books

  ```json
  
  { 
    _id: 1,
    title: "Book title One",
    categories: ["drama"],
    authors: [1, 2]
  }

  { 
    _id: 2,
    title: "Book title Two",
    categories: ["scifi"],
    authors: [1]
  }
  ```
Ex. 4 - Books and categories

Design a database to store data about books and their categories.

When accessing a book, also information on their categories must be retrieved.

When accessing a category, also information on its books must be retrieved.

• Books
  • Title
  • Categories
  • Authors

• Categories
  • Short name (i.e., tag)
  • Books
Books and categories (uneven N:M relationships)

• Categories
  {
    _id: 1,
    tag: "drama",
    books: [1, 2, 3, ...]
  }
  {
    _id: 2,
    tag: "scifi",
    books: [2]
  }

• Books (one-way embedding)
  {
    _id: 1,
    title: "Book title One",
    categories: [1],
    authors: [1, 2]
  }
  {
    _id: 2,
    title: "Book title Two",
    categories: [2],
    authors: [1]
  }