MongoDB Schema Design

Jon Tobin – Dir of Solutions Eng
@jontobs
Linkedin.in/JonathanETobin
MongoDB Overview

- NoSQL Document Oriented DB
- “Dynamic Schema”
- HA/DR/Sharding Built In
  - Election algorithm
  - Sharding engine
- Aggregation Framework
- “Developers Database”
  - Full driver library
  - Work outside of shell
- Easy to use
  - Read: “Easy to get started”
### Terms: What Do They Mean?

<table>
<thead>
<tr>
<th>MySQL</th>
<th>MongoDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database</td>
<td>Database</td>
</tr>
<tr>
<td>Table</td>
<td>Collection</td>
</tr>
<tr>
<td>Row</td>
<td>Document</td>
</tr>
<tr>
<td>Field</td>
<td>Key : value pairs</td>
</tr>
</tbody>
</table>
Practical Examples
Modeling Data - SQL

@ each intersection is a single scalar value
Modeling Data - SQL

Good
- Normalization gives guidelines
- JOIN to get data
- Minimizes redundancy
- Efficient updates
- Schema enforces data

TLDR: great for consistency and updates

Bad
- Three queries for data
- Pre-defined schema constrains agility
- Complex relationships

TLDR: querying and inserting can (will) be inefficient

WHY – It’s all about (co)location of relevant data
MongoDB Design Basics

- Known as a “Developers DB”
- No Joins
  - Joins are done in application
- Dynamic Schemas
  - Fields (keys) can be added anytime
  - Keys don’t need to be added to all docs (rows)
  - Keys can have multiple values or multiple sub-docs
- (De)Normalization is up to you
  - What best fits your application
  - Could be a mix
  - Anxiety inducing for RDBMS crowd
- 16MB BSON limit on docs
- Atomicity within a single document
JSON Types

- Number
- Text
- Boolean
- Array
- Object
- Null
Modeling Data - MongoDB

```json
{
    "_id" : ObjectId("507f1f77bcf86cd799439011"),
    "studentID" : 100,
    "firstName" : "Jonathan",
    "middleName" : "Eli",
    "lastName" : "Tobin",
    "classes" : [ 
        {
            "courseID" : "PHY101",
            "grade" : "B",
            "courseName" : "Physics 101",
            "credits" : 3
        },
        {
            "courseID" : "BUS101",
            "grade" : "B+",
            "courseName" : "Business 101",
            "credits" : 3
        }
    ]
}
```
QnD Doc Design Pointers

**Embed**
- Query performance priority
- Fields are fairly static
- Size of doc can be reasonably determined
- Eventual consistency acceptable

```json
{
    "_id" : ObjectId("53d98f1f..."),
    "firstName" : "Jonathan",
    "lastName" : "Tobin",
    "year" : 3,
    "classes" : [
        {
            "class" : "Calc 101",
            "credits" : 3,
        },
        {
        -etc-
    ]
}
```

**Reference**
- Insert performance priority
- Updates are common
- Immediate consistency necessary
- Field size can’t be determined

```json
{
    "_id" : ObjectId("53d98f1f..."),
    "firstName" : "Jonathan",
    "lastName" : "Tobin",
    "year" : 3,
    "classes" : [ ObjectID(<of_class_1>), ObjectID(<of_class_2>), ObjectID(<of_class_3>), ]
}
```
Embedding

- **Insert**
  - Quick
  - Semi efficient
- **Update**
  - studentID
    - Quick
  - courseID
    - Complex
    - Inefficient
    - Inconsistent
- **Query**
  - Fast
  - Efficient

Be mindful: cache thrashing

```javascript
Show collections
college.students

//sample document
{
  "_id" : ObjectId("507f1f77bcf86cd799439011"),
  "studentID" : 100,
  "firstName" : "Jonathan",
  "middleName" : "Eli",
  "lastName" : "Tobin",
  "classes" : [
    {
      "courseID" : "PHY101",
      "grade" : "B",
      "courseName" : "Physics 101",
      "credits" : 3
    },
    {
      "courseID" : "BUS101",
      "grade" : "B+",
      "courseName" : "Business 101",
      "credits" : 3
    }
  ]
}
```
Referencing

- Insert
  - Quick
  - Efficient
- Update
  - classes
    - Fairly quick
  - courseID
    - Efficient
    - Consistent
- Query
  - Fast
  - Efficient

Be mindful: join overhead

Show collections
  - college.students
  - college.courses
  - college.grades

//sample document
{
  "_id" : ObjectId("507f1f77bcf86cd799439011")
  "firstName" : "Jonathan",
  "lastName" : "Tobin",
  "year" : 3,
  "classes" : [
    ObjectId(<of_class_1>),
    ObjectId(<of_class_2>),
    ObjectId(<of_class_3>),
  ]
}
**MongoDB Design - QnD**

- **DEPENDS**: on use case
  - Embed
    - Efficient lookups
    - Infrequently changed data
    - Often queried data
    - Atomicity
  - Reference
    - Efficient writes
    - Oft excluded data (from queries)
    - Boundless additions
    - Doc size may approach 16MB limit
Finding Middle Ground

- Embed fields that are often fetched
  - If they don’t grow boundlessly
- Limit growing keys to 1/per doc
  - Move to last key
- Reference fields that are volatile
  - Or are occasionally queried
- Atomicity can be achieved @ single doc level
  - Take care in design
- Index judiciously
  - Re-evaluate often
- Store relevant data
  - Archive old data (when possible)
  - Or delete
Sharding
Sharded Cluster

- App Server
  - Router (mongos)
  - 1 or more Routers
- Config Servers (replica set)
- 2 or more Shards
  - Shard (replica set)
  - Shard (replica set)
Sharding

- Mongo distributes data based on shard key
  - Indexed single key
  - Indexed compound key
- Chunks data by key space
  - Range based
  - Hash based
- **Shard key is immutable**
- Balancing happens in background
  - Inside each shard
  - Between shards

Two distinct possibilities:
- Range: data has low entropy (scatter) > *key1 & key2 are likely to be together*
- Hash: data has high entropy (scatter) > *key1 & key2 are unlikely to be together*
Shard Keys

• For insert speed:
  • High-entropy shard key (mostly random).
  • Balances load across all shards.
  • Avoid migrations, can be expensive in MongoDB.
  • Range queries are scatter-gathers
  • “Scatter” is good.

• For query speed:
  • Low-entropy shard key (mostly sequential).
  • Range queries should only hit 1 shard.
    • Queries should include shard key.
  • “Scatter” is bad.
Thank You!

Please let us know what we can do to help

Please rate the talk via the Percona Live app