Design of No SQL / Mongo DB Databases

pm_jat @ daiict

Database Design

- What do you understand by Database Design?
- What is deliverable for “Database Design” as exercise?

Database Design - RDB

- What do you understand by Database Design in RDB?
  - Deciding “What Tables” we are going to have say
  - What Keys, Foreign Keys, and
  - other constraints on the database
- What is deliverable for “Database Design” as exercise?
  - A data model in the form of some diagram and document that describes Schema and other constraints
- Typical Design Process?
  - ER Model to Relational Model by applying known mapping rules, and then ensuring that relational are in desired normal form!

INPUT to Database Design process

- Conceptual Model:
  - Entities with attributes and primary keys
  - Relationships
    - Cardinality, and
    - Participation constraints
- Other constraints
  - Side Note: There are many constraints that are hard to be specified in schema, and therefore are implemented by programming: Triggers, Application Logic, UI etc.

Mapping from Conceptual Database to Implementation Database

- What is an Entity in database?
  - Entity, Entity Set, and Entity Type
  - Let a particular order be an entity, set of all orders be entity set, and schema of order be Entity Type!
- ER concepts to Relational Concepts

  Entity ==> Row

  Entity Set ==> Table

  Entity Type ==> Table Schema

Mapping from Conceptual Database to Implementation Database

- What is an Entity in database?
  - Entity, Entity Set, and Entity Type
  - Let a particular order be an entity, set of all orders be entity set, and schema of order be Entity Type!
- ER concepts to Document Database

  Entity ==> Document

  Entity Set ==> Document Collection

  Entity Type ==> roughly “Collection Level Schema” No SQL systems are on the stand of flexible schema!

Design of No-SQL databases

- Recall that No SQL database are said to be “Aggregation Oriented”!
- Aggregation oriented primarily means “embedded”; i.e., on embedded strategy rather than “normalized” strategy!
- Recall that: Aggregation Oriented databases serves two purposes

  (1) We store data together (embedded) that are queried together. This is crucial in cluster computing environment!

  (2) Programs also keeps data aggregated (in memory objects), and as a result we solve “impedance mismatch” problem!

Database Design – NO SQL

- When it comes to design of No SQL databases, we require figuring out “What Aggregations”?
- That means figuring out what entity is embedded in what, and what not?
- For instance, in Mongo DB Design, we require deciding
  - what all collections we need to have for given set of Entities (and Relationships), and
  - How relationships are to be represented through embedding or through referencing?

Database Design – NO SQL

- In another terms, we also say that in No SQL design process, we figure out what Aggregations, and how much "normalized"?
- Aggregation and Normalized are competing; therefore, often we require standing somewhere in between!
- "query load" plays an important role in determining a consensus between "aggregation" and "normalization" (in other words, "embedding" and "referencing")
- Let us see some of guiding principles in derived Design for Mongo DB (most principles discussed here apply to other no-sql systems too)

Design of Mongo DB Databases!

Let me share some thoughts from following two sources:

(1) Book: “MongoDB Applied Design Patterns” by Copeland, Rick.

(2) Rules of Thumb for MongoDB Schema Design: Part 1, 2, 3: https://www.mongodb.com/blog/post/6-rules-of-thumb-for-mongodb-schema-design-part-1

RDB and Normalization

- What is bad in following Table?

id	name	phone_numbers	zip_code
1	Rick	555-111-1234	30062
2	Mike	555-222-2345;555-212-2322	30062
3	Jenny	555-333-3456;555-334-3411	01209

RDB and Normalization

- What is bad in following Table?

id	name	phone_numbers	zip_code
1	Rick	555-111-1234	30062
2	Mike	555-222-2345;555-212-2322	30062
3	Jenny	555-333-3456;555-334-3411	01209

- Table is not in first normal form?
  What problem it has: “Update anomalies”

RDB and Normalization

- Normalized Tables

contact_id	name	zip_code
1	Rick	30062
2	Mike	30062
3	Jenny	01209

contact_id	phone_number
1	555-111-1234
2	555-222-2345
2	555-212-2322
3	555-333-3456
3	555-334-3411

Good and Bad about Normalized relations

- Good: “Update anomaly free”.
- Bad:
  - Requires JOINs; and joins are expensive to execute even in non-distributed environment.
  - In distributed environments JOINs are forbiddingly expensive!
- Therefore often we have ‘de-normalized’ table!
- “Denormalization for performance” is quiet norm in modern databases!

Normalization and Mongo DB

- Normalization is for relational databases, and have understood well in that context.
- No SQL systems have been designed from scratch, therefore the term “Normalization” may also require redefining for No-SQL!
- Let us say using less of “aggregation” is called as normalized design.

Normalization and Problems in MongoDB

- Consider same contacts DB
- Here is unnormalized in relational sense?

```
{
  "_id": 3,
  "name": "Jenny",
  "zip_code": "01209",
  "numbers": [ "555-333-3456", "555-334-3411" ]
}
```

Normalization and Problems in MongoDB

- Now normalized (two separate collections).
- While such a design is favour in relational systems, it may not be so in No SQL systems. We will in a short while!

```
// Contact document:
{
  "_id": 3,
  "name": "Jenny",
  "zip_code": "01209"
}

// Number documents:
{ "contact_id": 3, "number": "555-333-3456" }
{ "contact_id": 3, "number": "555-334-3411" }
```

NO SQL Database Design Considerations

- “Embedding” and “Normalized” are two competing stands we have here.
- Let us which is good for what?
  - Embedding for Locality and Performance
  - Embedding for Atomicity and Isolation
  - Referencing (Normalized) for Flexibility
  - Referencing (Normalized) for Potentially High-Arity Relationships
  - Referencing (Normalized) for implementing “Many-to-Many Relationships”

Why Embedding?

Embedding for Locality and Performance

- **It is well known that Normalized design suffers from performance**!
- Requires Joining.
- Join becomes more cause of concern when we require to have Distributed JOINs.
- Example:

Why Embedding?

```
// Contact document:
{
  "_id": 3,
  "name": "Jenny",
  "zip_code": "01209"
}

// Number documents:
{ "contact_id": 3, "number": "555-333-3456" }
{ "contact_id": 3, "number": "555-334-3411" }
```

- Normalized:

  contact_info = db.contacts.find_one({'_id': 3}) ;

  contact_numbers = db.numbers.find({'contact_id': 3});

- Denormalized:

  contact_info = db.contacts.find_one({'_id': 3})

De-normalized (embedded) design

- Contacts!

```
{
  "_id": 3,
  "name": "Jenny",
  "zip_code": "01209",
  "numbers": [ "555-333-3456", "555-334-3411" ]
}
```

Why Embedding?

Embedding for Atomicity and Isolation

- In normalized design an “aggregation” is split into multiple documents, and stored in different collections.
- In such a case operations on aggregation requires to be performed on multiple collections.
- Ensuring “atomicity” in normalized case is hard in Distributed environment!
- Ensuring “atomicity” for multiple document transactions is hard!
- Mongo DB provides “Document Level” atomicity only!

Why Embedding?

```
// Contact document:
{
  "_id": 3,
  "name": "Jenny",
  "zip_code": "01209"
}

// Number documents:
{ "contact_id": 3, "number": "555-333-3456" }
{ "contact_id": 3, "number": "555-334-3411" }
```

- Normalized:

```
db.contacts.remove({'_id': 3})
db.numbers.remove({'contact_id': 3})
```

- Denormalized:

```
db.contacts.remove({'_id': 3}); BIG YES
```

Why Referencing?

Referencing for Flexibility

- Embedding turns out to be complex when embedded entity is independently queried
- Consider two entities in a blogging application “posts” and ”comments”, and (1:N)
- Let us say we have a single collection “posts” where “comments” are embedded in “posts”, a sample document is

```
{
  "_id": "First Post",
  "author": "Rick",
  "text": "This is my first post",
  "comments": [
     { "author": "Stuart", "text": "Nice post!" },
     ...
     ]
}
```

Referencing for Flexibility

```
{
   "_id": "First Post",
   "author": "Rick",
   "text": "This is my first post",
   "comments": [
      { "author": "Stuart", "text": "Nice post!" },
      ...
      ]
}
```

- And let us say we submit following query, with the most guessed objective?

```
db.posts.find(
     {'comments.author': 'Stuart'},
     {'comments': 1})
```

Referencing for Flexibility

- Following could be typ

```
db.posts.find(
    {'comments.author': 'Stuart'},
    {'comments': 1})
```

```
{ "_id": "First Post",
  "comments": [
    { "author": "Stuart", "text": "Nice post!" },
    { "author": "Mark", "text": "Dislike!" } ] },
{ "_id": "Second Post",
  "comments": [
    { "author": "Danielle", "text": "I am intrigued" },
    { "author": "Stuart", "text": "I would like to subscribe" } ] }
```

- Hope, you see the problem in the result?
- It is giving all documents that have comments from 'Stuart' where as we required only comments from 'Stuart'!

Referencing for Flexibility

Normalized storage will help us in getting more relevant results!

```
// db.posts schema
{
  "_id": "First Post",
  "author": "Rick",
  "text": "This is my first post"
}

// db.comments schema
{
  "_id": ObjectId(...),
  "post_id": "First Post",
  "author": "Stuart",
  "text": "Nice post!"
}
```

```
db.comments.find({"author": "Stuart"})
```

Why Referencing?

Referencing for Potentially High-Arity Relationships

- When arity is high, it may not be advisable to embed
- If degree of embedded entity is very high. Say a very popular posts have several thousand comments.
- In this “embedding” becomes expensive in following terms:
  - The document will larger and hence using large RAM when loaded
  - Large document shall also slow the update performance
  - MongoDB documents have a hard size limit of 16 MB.
- So in all these situations, “referencing” may be chosen!

Why Referencing?

Referencing for Potentially High-Arity Relationships

- When document size becomes large, “referencing” may be preferred.
- This do make sense also, in blogging site, we may not be requiring to read all comments for a post, only few are read and shown!

"Many-to-Many Relationships"

- Suppose we have two entities "products" and "categories" in an ecommerce application.
- A product can be in many categories and obviously a category has many products ==> **Many to Many** cardinality!
- From relation relational learning, and favouring "referring" we may decide to having following three collections:
  - "products", "categories", "product_category"

"Many-to-Many Relationships"

- Retrieving a category with its products in this solution is complex.

  {1, 101, C1}

  {2, 101, C5}

  {3, 102, C1}

```
// db.product schema
{ "_id": "My Product", ... }

// db.category schema
{ "_id": "My Category", ... }

// db.product_category schema
{ "_id": ObjectId(...),
  "product_id": "My Product",
  "category_id": "My Category" }
```

"Many-to-Many Relationships"

- Retrieving a category with its products in this solution is complex.

```
// db.product schema
{ "_id": "My Product", ... }

// db.category schema
{ "_id": "My Category", ... }

// db.product_category schema
{ "_id": ObjectId(...),
  "product_id": "My Product",
  "category_id": "My Category" }
```

Queries for getting details of product (id=123) and its categories

```
product = db.product.find_one({"_id": "123"})
category_ids = db.product_category.find( { "product_id": "123" })
categories = db.category.find({"_id": { "$in": category_ids } })
```

“Many-to-Many Relationships”

in one

```
// db.product schema
{ "_id": "My Product",
  "categories": [
      { "_id": "My Category", ... }
      ...] }
```

READ is very efficient here

```
db.product.find_one({"_id": product_id})
```

```
// db.category schema
{ "_id": "My Category",
  "products": [
    { "_id": "My Product", ... }
    ...] }
```

Polymorphic Schema

- Storing inheritance hierarchy has been fundamentally foreign to relational databases
- No-SQL database's characteristics of "flexible schema" helps in storing entities from a sub-class hierarchy!
- With this schemeless-ness characteristics of Mongo, we can store Books, Movies, Mobiles, in a same "document collection"
- Product(prod_id, name, supp_id, cost): Books, Movies, Mobiles

Six Rule of Thumbs for Mongo DB Design from [2]

- One: favour embedding unless there is a compelling reason not to
- Two: needing to access an object on its own is a compelling reason not to embed it
- Let "needing to access an object on its own" be reason of not embedding
- Example #1: Already seen "Blog Post" and "Comments", and we require querying comments from a given user?
- Example #2: Product and Parts (that are used in the Product), we require query part independent of products in which they are used

Six Rule of Thumbs for Mongo DB Design from [2]

- Three: Arrays should not grow without bound. SIZE comes on the way (already seen concerns of bigger size of document)
- Arrays are used for storing objects or references in “one side”
- If there are more than a couple of hundred documents on the “many” side, don’t embed them; [
- if there are more than a few thousand documents on the “many” side, do not even store references in one side

Six Rule of Thumbs for Mongo DB Design from [2]

- In relation to embedding, the articles classifies one to many cardinality in following three
- Here are solution when we want store relationships in N side!
  - One to Few (Example: Person and Addresses)
    [can embed] [embed or not that is separate question]
  - One to Many (Example: Product and Parts)
    [No embedding, store references]
  - One to Squillions (Example: Host and LogMessages)
    [Not even reference embedding] [Have reference in many side only, parent referencing]

Rule of Thumbs for Mongo DB Design

- Four: Don’t be afraid of application-level joins.
- What is application level joins?
- Get product from products, get Part IDs from embedded array in product document, fetch part documents for IDs in the array!

```
> product = db.products.findOne({catalog_number: 1234});
//Fetch all the Parts that are linked to this Product
> product_parts = db.parts.find({_id:{$in:product.parts}
}).toArray();
```

Rule of Thumbs for Mongo DB Design

- Four: Don’t be afraid of application-level joins.
- if correct index is available, then application-level joins are barely more expensive than server-side joins in a relational database.

Rule of Thumbs for Mongo DB Design

- Five: Consider the write/read ratio when de-normalizing (embedding)
- A field that will mostly be read and only seldom updated is a good candidate for denormalization
- if we de-normalize a field that is updated frequently then the extra work of finding and updating all the instances is likely to overwhelm the savings that you get from de-normalizing.
- For instance if part is used in multiple products, and updating part in one product requires updating it in all other products!

Rule of Thumbs for Mongo DB Design

- Six: As always with MongoDB, how you model your data depends – entirely – on your particular application's data access patterns.
- You want to structure your data to match the ways that your application queries and updates it.

Summarized

- Embed or Not? If you need access the object on the “N” side separately, or only in the context of the parent object?
- Embed Which side? Entity that more often queried?
- While embedding in N side decide according to arity of cardinality: “one-to-few”, “one-to-many”, or “one-to-squillions”?
- If updates are more than reads, embedding may turn out to be expensive

Summarized

- Embedding can occur in Many side too; Publisher information getting stored in every Book. Often “suffers from anomalies”
- Reference embedding in many side is a choice when we have “one-to-squillions” relationship.

References/Further Reading

[1] Copeland, Rick. *MongoDB Applied Design Patterns: Practical Use Cases with the Leading NoSQL Database*. " O'Reilly Media, Inc.", 2013.

[2] Rules of Thumb for MongoDB Schema Design: Part 1, 2, and 3
https://www.mongodb.com/blog/post/6-rules-of-thumb-for-mongodb-schema-design-part-1