Designing Document Database Models

Daniel Zappala

CS 360 Internet Programming
Brigham Young University
Document Databases
Document Databases

- store key/value pairs, and the value is generally a JSON document

```json
{
  '_id': <ObjectId>,
  'version': '2.7',
  'user_id': 'KWCH-G25',
  'api_id': 'KL1S-Z4L',
  'api_data': {...},
  'name': {'given': 'Carmelo', 'full': 'Carmelo Zappal\xe0', 'family': 'Zappal\xe0'},
  'gender': 'Male',
  'living': 'False',
  'facts': {'death': {'date': '16 March 1983', 'year': '1983', 'place': 'Burbank, Los Angeles, California, United States', 'day': '16', 'month': '3'}, 'birth': {'date': '6 October 1886', 'year': '1886', 'place': 'Linguaglossa, Catania, Sicily, Italy', 'day': '6', 'month': '10'}}
...}
```
Data Models

- **embedded**
  - store related information in a single document
  - useful for one-to-one relationships
  - useful for one-to-many relationships when the child documents always appear with the parent, e.g. comments on a blog

- **normalized**
  - store related information across separate documents
  - use references to create relationships
  - useful for one-to-many relationships when the children are accessed separately, e.g. want to list a particular user's comments on all blog entries
  - useful for many-to-many relationships
  - useful for very large hierarchical datasets, where the size of a document would exceed database limits
PyMongo Example

```python
from pymongo import MongoClient
client = MongoClient()

// get a database
db = client.blog

// get a collection
posts = db.posts

// create a post
import datetime
post = {
  "author": "Mike",
  "text": "My first blog post!",
  "tags": ["mongodb", "python", "pymongo"],
  "date": datetime.datetime.utcnow()
}

// insert
posts.insert(post)

// get all posts
posts.find()

// get posts by Mike
posts.find({"author": "Mike"})
```
Why Use a Document Database?

- **scale**
  - cloud computing provides cheap storage, ability to easily add more servers
  - but data must be spread across multiple servers, difficult to use SQL joins in across distributed set of tables

- **unstructured data**
  - social media, multimedia
  - SQL is best suited for structured data

- **agile development**
  - database scheme must rapidly evolve
  - easier to do when all items in a table don’t need to maintain consistent structure
Document Databases are Good At...

- independent documents
  - fast read performance
  - easy to distribute across servers
- easy application logic
  - object-oriented syntax translates directly into document storage
- unstructured data
  - documents can store whatever keys and values the app requires
  - migrations not needed in advance, can adapt on the fly
Consistency

• CAP Theorem
  • a database can only have two of Consistency, Availability, and Partition tolerance
  • under Partition, a system cannot maintain both Availability and Consistency
• strongly consistent
  • favor consistency over availability
  • relational database
  • MongoDB
• eventually consistent
  • favor availability over consistency
  • Amazon Dynamo, CouchDB, Riak

• easier to provide even load distribution and multi-data center support in eventually consistent systems
• easier to write code for strongly consistent systems, applicable to a wider set of problems
Replication

![Diagram showing replication in a document database system with a primary and secondary nodes.]
Heartbeats
Failover

Election for New Primary

Secondary ⇄ Heartbeat ⇄ Secondary

New Primary Elected

Primary ⇄ Replication ⇄ Heartbeat ⇄ Secondary
Sharding

Collection

Shard A

Shard B

Shard C

Shard D

256 GB

256 GB

256 GB

256 GB
Sharding Configuration

- App Server
  - Router (mongos)
- 2 or more Routers
- 3 Config Servers
- 2 or more Shards
  - Shard (replica set)
  - Shard (replica set)
How Riak Works
Storage

- all nodes in a Riak cluster are equivalent; there is no master
- data is distributed across nodes using consistent hashing
  - stores key/value pairs in a bucket namespace (roughly a table)
  - hashes bucket/key combination into a 160-bit integer space
  - each node responsible for portions of the space
- 4-node cluster with 32 partitions:

![Diagram of a ring with 32 partitions and nodes](image)
Replication

- writes specify a replication number $W$, specifying amount of replication
- stored in that number of consecutive locations
- nearby virtual locations map to physical locations that may be geographically distinct

```
put("artist", "REM")
```

(N=3)
Eventual Consistency

• reads specify a level of consistency, $R$
  • specifies how many replicas must return results for a successful read
  • send a request to all nodes where data is stored, return when $r$ have responded
• $W + R > N$ ensures strict consistency
• $W + R \leq N$ provides eventual consistency for better availability or lower latency
• a write may occur during a partition
  • reads will return different values in each partition
  • when the partition is healed, write ordering will eventually result in consistent data
Example
Example

- Sample FamilyTree App model
- uses PyMongo