NoSQL Data Storage

Big Data Management

Vytautas Tumas(vt50)
Software Engineering Year 4

February 21, 2016

Contents

1 Database Management System

For this coursework I chose MongoDB DBMS. MongoDB falls into the NoSQL database category, a single database holds a set of collections and each collection contains a group of documents. Collections are used to store related data, they are the equivalent of a table in RDBMS, however a collection does not enforce a schema. A document is a set of key-value pairs, because MongoDB uses dynamic schemas, the documents in the collection are not required to have the same keys.

MongoDB is easily scalable system. Unlike traditional RDBMS where the way to improve the performance is to upgrade the machine the database is running, MongoDB scales out. It uses a process called sharding to distribute data across commodity hardware. Each shard of the database is able to run on a separate machine, thus to cope with the data growth, additional machines will have to be added to the database network. Each shard is an independent database, a collection of shards makes up a single logical database.

2 Data Model

Figure 1 shows the model of the new database. Unlike in the traditional model, the *movies* schema has been updated to hold the genre information of the particular movie. The main motivation behind this, is that the genre of a movie and the name of the genre will not change, thus there is no need to store these in a separate collection. However, I have discovered that querying for movies that have multiple genres is more complicated than in a RDBMS model, the queries require multiple **$in** operators aggregated with the **$and** operator. The *legacy_id* field was added to help with the migration of the *ratings* table. The only change to the *users* schema is the added *legacy_id* field to help with the migration of the *ratings* table.

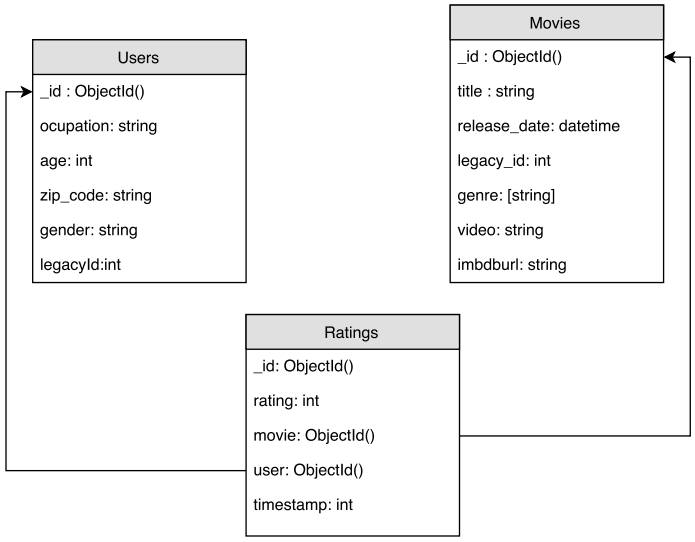

Figure 1: MongoDB schema model

The main complexity of the RDBMS model is the relationship in the ratings model, between the user and the movie. I explored three ways to store the ratings data.

For the first schema, I planned to have a list of ratings for each movie embedded into the movie document, however this schema was flawed. Firstly, every time a movie is rated, the document for the particular movie will have to be updated. Because the update mechanism is implemented on the users side and NoSQL database do not provide field constraints for embedded documents, a single user could potentially rate a movie more than once. The second drawback is that a single document can be upto 16MB in size. As the result only a limited number of ratings can be stored in the document. Finally, assuming it is possible to overcome the 16MB document size constraint, a problem will occur when sharding the database. It is possible to split a collection over multiple shards, however it is not possible to split a single document over multiple shards. Therefore, the more ratings the single movie has, the longer it will take to query it. The second schema, is similar to the first, but instead of storing the ratings embedded in the movie document, store user rating embedded in the user document. In addition to the issues which apply to the first schema, now when ever we want to query the rating of a single movie, we will have to retrieve every movie document to check whether the user has rated the movie, this is an unnecessary level of complexity, which can be avoided by storing the ratings in a separate collection.

By storing the movie ratings data in a separate collection we can add a *unique compound index* on the movie and user ids to ensure that a user can rate a movie only once. Because each document in the ratings collection represents a single rating we don't have to worry about the size of one document. Lastly, because the ratings are stored in a separate collection, the collection can be easily shared and the overall performance improved.

3 Migration

To migrate the data from MySQL to MongoDB I used the *peewee* library to access the MySQL database and the *pymongo* library to access the MongoDB. The *peewee* library provides the tools to generate a class for the each database table (Code 3), thus I could easily query the MySQL database without worrying about issues with encoding. The users table (Code 3) was migrated by selecting all the users from the movielens database, adding a new *legacyId* field for reference, and writing the user to the MongoDB *users* collection. The movies table (Code 3)

was migrated in a similar way, to make future querying by date easier, the *release_date* field was converted to *datetime* representation. To migrate the ratings collection (Code 3), we first select all the movies, then iterate the list finding the ratings for the movie. For each rating, we find the related user, update the rating document with the new *id* of the movie and the user and write the new document to the collection. The migration of the users and movies table took around 30 seconds each as these tables 943 and 1682 entries respectively. The movie ratings table took around 10 minutes to migrate, this is because for each movie rating we have to query the related user, as the result the same user is likely to be fetched more than once. This problem can be solved by introducing a cache for users. Before fetching the user information from the database, the script would check a local dictionary of $< id, user >$ pairs, if the user is not in the cache, the information is then fetched from the database and added to the cache.

Appendices

Class representation of movielens tables.

```
from peewee import *

database = MySQLDatabase('movielens', **{'user': 'vt50', 'password': 'abcvt50354',

class UnknownField(object):
    pass

class BaseModel(Model):
    class Meta:
        database = database

class Genres(BaseModel):
    genre = CharField(null=True)

    class Meta:
        db_table = 'genres'

class Movies(BaseModel):
    imdburl = CharField(db_column='IMDBURL', null=True)
    release_date = CharField(null=True)
    title = CharField(null=True)
    video = CharField(null=True)

    class Meta:
        db_table = 'movies'

class MovieGenres(BaseModel):
    genre = ForeignKeyField(db_column='genre', rel_model=Genres, to_field='id')
```

```
    movie = ForeignKeyField(db_column='movie', rel_model=Movies, to_field='id', rela

    class Meta:
        db_table = 'movie_genres'
        indexes = (
            (('movie', 'genre'), True),
        )
        primary_key = CompositeKey('genre', 'movie')

class Users(BaseModel):
    age = IntegerField(null=True)
    gender = CharField(null=True)
    occupation = CharField(null=True)
    zip_code = CharField(null=True)

    class Meta:
        db_table = 'users'

class Ratings(BaseModel):
    movie = ForeignKeyField(db_column='movie', rel_model=Movies, to_field='id', rela
    rating = IntegerField(null=True)
    timestamp = IntegerField(null=True)
    user = ForeignKeyField(db_column='user', rel_model=Users, to_field='id')

    class Meta:
        db_table = 'ratings'
        indexes = (
            (('user', 'movie'), True),
        )
        primary_key = CompositeKey('movie', 'user')
```

User table migration

```
def migrate_user():
  drop_collection(USER)
  count = 0
  for user in Users.select():
    userDict = model_to_dict(user)
    userDict['legacyId'] = userDict.pop('id', None)
    insert_to_mongo(userDict, USER)
    count += 1
  print("Users migrated " + str(count))
```

Movie table migration

```
def migrate_movie():
  drop_collection(MOVIE)
  count = 0
  for movie in Movies.select():
    movieDict = model_to_dict(movie)
    #init the genre list
    movieDict['genre'] = []
    # remove the id field, mongo will generate it's own but keep the legacyId
    movieDict['legacyId'] = movieDict.pop('id', None)
    try:
      movieDict['release_date'] = datetime.strptime(movieDict['release_date'], '%d-%
    except:
      date = get_movie_date(movieDict['title'])
```

```
      if date is not None:
        movieDict['release_date'] = datetime.datetime(int(date), 1, 1, 0, 0)
      else:
        movieDict['release_date'] = None
    # add related genres
    for genre in movie.genres:
      movieDict['genre'] += [genre.genre.genre]
    insert_to_mongo(movieDict, MOVIE)
    count += 1
  print("Movies migrated " + str(count))
```

Movie rating migration

```
def migrate_rating():
  drop_collection(RATING)
  count = 0
  movies = get_collection_items(MOVIE)
  for movie in movies:
    dicts = []
    # find all ratings for the movie
    ratings = Ratings.select().where(Ratings.movie == movie['legacyId'])
    for rating in ratings:
      ratingDict = model_to_dict(rating)
      ratingDict['legacyId'] = ratingDict.pop('id', None)
      # find the user who gave the rating
      user = find_item(USER, 'legacyId', rating.user.id)
      if user is not None:
        ratingDict['user'] = ObjectId(str(user['_id']))
        ratingDict['movie'] = ObjectId(str(movie['_id']))
        dicts.append(ratingDict.copy())
        count += 1
      else:
        print("User not found")
    insert_many(RATING, dicts)
    print("Count: " + str(count))
  print("Ratings migrated " + str(count))
```