Indholdsfortegnelse

[Introduktion 2](#_Toc342652099)

[Find the ID of the film "The Rock" from 1996? (Use find()): 2](#_Toc342652100)

[Find how many 5-star ratings "The Rock" has received received? (Use find()) 2](#_Toc342652101)

[Calculate a document/cursor describing the distribution of ratings of The Rock? (That is, how many 1, 2, 3, 4, 5 star ratings?) (Use mapReduce()) 2](#_Toc342652102)

[Calculate the average rating of The Rock? (Use mapReduce(). You may calculate it on either the original ratings collection or on the output collection from the previous mapReduce.) [I get 3.723] 2](#_Toc342652103)

[Count how many movies are classifed as 'Animation' movies? 3](#_Toc342652104)

[Generate a collection "drama" containing all titles of movies classified as genre Drama. 3](#_Toc342652105)

[How many users have profession 'writer' (occupation=20)? 3](#_Toc342652106)

[What is the distribution of ratings for The Rock given by writers? (Tricky :) 3](#_Toc342652107)

[The MovieLens data is obviously normalized and thus the worst possible format for a document-based NoSQL database. Design a new 'schema' in JSON that would be much more effecient for Mongo. 5](#_Toc342652108)

[Hand-craft a small set of documents (5-10) using your new schema which examplify MovieLens data, put them into a new collection, and redesign the above queries so they operate on the new schema. 5](#_Toc342652109)

[Optional: Design a (set of) map-reduce functions that will translate the three MovieLens collections into a single collection with your newly defined schema (denormalization). (Tricky, I haven't been able to do it (yet) :) 5](#_Toc342652110)

[Optional: Use two/more machines to explore the auto-sharding facilities in MongoDB. 5](#_Toc342652111)

[Optional: Use three machines to explore the replication facilities in MongoDB. 6](#_Toc342652112)

[Optional: Read the MovieLens data into a SQL database of your choice and compare performance between SELECTS and Map-Reduce. (Postings on the web generally state that Map Reduce is slow in Mongo.) 6](#_Toc342652113)

# Introduktion

Dette dokument indeholder besvarelsen på opgave H6 i kurset Software Arkitektur i Praksis.

# Find the ID of the film "The Rock" from 1996? (Use find()):

|  |
| --- |
| db.movies.find({ title: "The Rock" },{\_id: 1}) |

No such movie found.

A little searching using:

|  |
| --- |
| db.movies.find({ title: /Rock/ }) |

revelas a “Rock, The (1996)”, and inserting this we get:

|  |
| --- |
| db.movies.find({ title: "Rock, The (1996)" },{\_id: 1}) |

id = 733

# Find how many 5-star ratings "The Rock" has received received? (Use find())

|  |
| --- |
| db.ratings.count({ rating: 5, movie\_id: 733 }) |

= 299

# Calculate a document/cursor describing the distribution of ratings of The Rock? (That is, how many 1, 2, 3, 4, 5 star ratings?) (Use mapReduce())

|  |
| --- |
| var mapFunction1 = function() {  var value = 1;  emit(this.rating, value);  };  var reduceFunction1 = function(key, values) {  return Array.sum(values);  };  db.ratings.mapReduce(  mapFunction1,  reduceFunction1,  { out: "ratings\_distribution", query: { movie\_id: 733 } }  ) |

# Calculate the average rating of The Rock? (Use mapReduce(). You may calculate it on either the original ratings collection or on the output collection from the previous mapReduce.) [I get 3.723]

|  |
| --- |
| var mapFunction1 = function() {  emit(this.movie\_id, this.rating);  };  var reduceFunction1 = function(key, values) {  return Array.sum(values) / values.length;  };  db.ratings.mapReduce(  mapFunction1,  reduceFunction1,  { out: { inline: 1 }, query: { movie\_id: 733 } }  ) |

= 3.6444281524926687

Unfortunately this is not correct. The error should be found in the reduce function. As there is only one key it is expected that the reduce function is only called once. This is not the case. It is possible that the reduce function is called multiple times, the first time reducing part of the result, and the second time with the reduced result along with the remaining data. We know there is only two calls due to the output from mapreduce.

|  |
| --- |
| {  "results" : [  {  "\_id" : 733,  "value" : 341  }  ],  "timeMillis" : 1346,  "counts" : {  "input" : 1340,  "emit" : 1340,  "reduce" : 2,  "output" : 1  },  "ok" : 1,  } |

There is a finalize method which may be called at the very end, which could be usable. To have something to compare with here is the C-solution to the problem:

|  |
| --- |
| var x = db.ratings.find({ movie\_id: 733 }, { \_id: 0, rating: 1} )  var ratingSum = 0;  for (var i = 0; i < x.length(); ++i) {  ratingSum += x[i].rating;  }  ratingSum / x.length() |

= 3.723

Using mapReduce with finalize we get the following

|  |
| --- |
| var mapFunction1 = function() {  var v = { count: 1, sum: this.rating };  emit(this.movie\_id, v);  };  var reduceFunction1 = function(key, values) {  reduceVal = { count: 0, sum: 0 };  for (var i = 0; i < values.length; ++i) {  reduceVal.count += values[i].count;  reduceVal.sum += values[i].sum;  }  return reduceVal;  };  var finalizeFunction1 = function(key, reducedValue) {  return reducedValue.sum / reducedValue.count;  }  db.ratings.mapReduce(  mapFunction1,  reduceFunction1,  {  out: { inline: 1 },  query: { movie\_id: 733 },  finalize: finalizeFunction1  }  ) |

= 3.723134328358209

What can we learn from this? We have a hypothesis. The Method reduce is called in a very special way, which is important to know. It is best exemplified with a sample

|  |
| --- |
| Input sample (key,value):  { { 1, 27 },{ 1,12 },{1,10},{2,23} } |

Emit is called for each key,value pair, and in this example it simply returns the same key,value pair. Reduce is then called for each key given the matching values. However it may be called multiple times for each key.

Our Reduce performs a strcat of the values.

Here is an example of a call pattern:

|  |
| --- |
| Call 1:  Reduce(1, { 27, 12 }) => return “2712”;  Call 2:  Reduce(2, { 23 }) => return “23”;  Call 3:  Reduce(1, {“2712”, 10 }) => return “271210”; |

In the end a finalize call is performed for each key with the result of the final reduce for the given key.

For our strcat example it works, but what if we try an average.

|  |
| --- |
| Call 1:  Reduce(1, { 27, 12 }) => return (27 + 12) / 2 = 19.5;  Call 2:  Reduce(2, { 23 }) => return 23 / 1 = 23;  Call 3:  Reduce(1, {19.5, 10 }) => return (19.5 + 10) / 2 = 14.75; |

But the true value is (27 + 12 + 10) / 3 = 16.33

It is therefore extremely important to use the reduce function correctly.

# Count how many movies are classifed as 'Animation' movies?

|  |
| --- |
| db.movies.count( { genres: { $all: [ "Animation" ] } } ) |

= 205

# Generate a collection "drama" containing all titles of movies classified as genre Drama.

|  |
| --- |
| var x = db.movies.find( { genres: { $all: [ "Drama" ] } }, { \_id: 0,title: 1 } )  for (var i = 0; i < x.length(); ++i) {  db.drama\_movies.insert( { movie : x[i] });  }  // Alternative using mapReduce  var mapFunction1 = function() {  emit(this.\_id, this.title);  };  var reduceFunction1 = function(key, values) {  return values;  };  db.movies.mapReduce(  mapFunction1,  reduceFunction1,  { out: "drama\_movies", query: { genres: { $all: [ "Drama" ] } } }  ) |

# How many users have profession 'writer' (occupation=20)?

|  |
| --- |
| db.users.count({ occupation: 20 }) |

= 281

# What is the distribution of ratings for The Rock given by writers? (Tricky :)

This is indeed tricky. The simplest – use standard logic notation and variables.

|  |
| --- |
| var x = db.users.find({ occupation: 20 },{\_id:1})  var mergeUsersAndRatings = function() {  var counts = { };  var values = { };  for (var k = 1; k < 6; ++k) {  values["rating." + k] = 0;  values["count." + k] = 0;  }    var x = db.users.find({ occupation: 20 },{\_id:1})  var y = db.ratings.find( { movie\_id: 733 }, { \_id:0,user\_id:1,rating:1 } )  var n = 0;  for (var i = 0; i < y.length(); ++i) {  var found = 0;  for (var j = 0; j < x.length() && found == 0; ++j) {  if (y[i].user\_id == x[j].\_id) {  found = 1;  var rating = y[i].rating;  values["rating." + rating] += rating;  values["count." + rating] += 1;  ++n;  }  }  }  for (var k = 1; k < 6; ++k) {  values["average." + k] = values["rating." + k] / values["count." + k];  }  return values;  }  mergeUsersAndRatings(); |

Not a very efficient method, but it works.

An alternative is the mapReduce, which we need to use to make a “merge”. Merge is not supported in Mongo, so we have to use the merging capabilities of mapReduce.

|  |
| --- |
| var mapFunctionUsers = function() {  var mergeData = {  user\_id: this.\_id,  occupation: this.occupation,  is\_rating: 0,  ratings: [0,0,0,0,0]  }  emit(this.\_id, mergeData);  };  var mapFunctionRatings = function() {  var mergeData = {  user\_id: this.user\_id,  occupation: 0,  is\_rating: 1,  ratings: [0,0,0,0,0]  }  mergeData.ratings[this.rating - 1] = 1;  emit(this.user\_id, mergeData);  };  var reduceFunction = function(key, values) {  var mergeData = {  user\_id: 0,  occupation: 0,  is\_rating: 1,  ratings: [0,0,0,0,0]  }  for (var i = 0; i < values.length; ++i) {  if (values[i].is\_rating == 1) {  for (var j = 0; j < 5; ++j) {  mergeData.ratings[j] += values[i].ratings[j];  }  }  else {  mergeData.occupation = values[i].occupation;  }  }  mergeData.user\_id = key;  return mergeData;  };  db.users.mapReduce(  mapFunctionUsers,  reduceFunction,  { out: { reduce: "user\_rating" } }  )  db.ratings.mapReduce(  mapFunctionRatings,  reduceFunction,  { out: { reduce: "user\_rating" } }  ) |

This results in a single collection containing the merge of users and ratings, as shown below.

|  |
| --- |
| { "\_id" : 15, "value" : { "user\_id" : 15, "occupation" : 7, "is\_rating" : 1, "ratings" : [ 4, 30, 68, 95, 4 ] } }  { "\_id" : 16, "value" : { "user\_id" : 16, "occupation" : 0, "is\_rating" : 1, "ratings" : [ 4, 9, 9, 8, 5 ] } }  { "\_id" : 17, "value" : { "user\_id" : 17, "occupation" : 1, "is\_rating" : 1, "ratings" : [ 0, 9, 35, 98, 69 ] } }  { "\_id" : 18, "value" : { "user\_id" : 18, "occupation" : 3, "is\_rating" : 1, "ratings" : [ 41, 19, 51, 89, 105 ] } }  { "\_id" : 19, "value" : { "user\_id" : 19, "occupation" : 10, "is\_rating" : 1, "ratings" : [ 11, 30, 71, 88, 55 ] } }  { "\_id" : 20, "value" : { "user\_id" : 20, "occupation" : 14, "is\_rating" : 1, "ratings" : [ 0, 1, 4, 11, 8 ] } }  has more |

It is possible to query this directly with: db.user\_rating.find( { 'value.occupation': 20 } )

|  |
| --- |
| { "\_id" : 356, "value" : { "user\_id" : 356, "occupation" : 20, "is\_rating" : 1, "ratings" : [ 0, 0, 2, 9, 10 ] } }  { "\_id" : 362, "value" : { "user\_id" : 362, "occupation" : 20, "is\_rating" : 1, "ratings" : [ 3, 9, 20, 15, 8 ] } }  { "\_id" : 382, "value" : { "user\_id" : 382, "occupation" : 20, "is\_rating" : 1, "ratings" : [ 1, 5, 11, 14, 25 ] } }  { "\_id" : 406, "value" : { "user\_id" : 406, "occupation" : 20, "is\_rating" : 1, "ratings" : [ 10, 18, 47, 61, 11 ] } }  { "\_id" : 454, "value" : { "user\_id" : 454, "occupation" : 20, "is\_rating" : 1, "ratings" : [ 61, 35, 37, 51, 82 ] } }  has more |

To extract the average of the users with occupation 20 we use the mapReduce again.

|  |
| --- |
| var mapFunction1 = function() {  var ratingData = {  ratings : this.value.ratings;  }  emit(this.value.occupation, ratingData);  };  var reduceFunction1 = function(key, values) {  var ratingData = {  ratings: [0,0,0,0,0]  }  for (var i = 0; i < values.length; ++i) {  for (var j = 0; j < 5; ++j) {  ratingData.ratings[j] += values[i].ratings[j];  }  }  return ratingData;  };  db.user\_rating.mapReduce(  mapFunction1,  reduceFunction1,  { out: "rating\_distribution", query: { 'value.occupation': 20 } }  ) |

Unfortunately this does not work!!!

# The MovieLens data is obviously normalized and thus the worst possible format for a document-based NoSQL database. Design a new 'schema' in JSON that would be much more effecient for Mongo.

I Mongo er der naturligvis ingen skemaer ☺, men for at gøre det sår effektivt så muligt i Mongo skal man have det hele i en collection.

|  |
| --- |
| Ratings: { "\_id", "movie\_id" : 733, "user\_id" : 1, "rating" : 3 }  Movies: { " \_id" : 20, "title" : "Money Train (1995)", "genres" : "Action" }  Users: { "\_id" : 20, "gender" : "M", "age" : 25, "occupation" : 14, "zip\_code" : "55113" }  Combined to:  { "\_id", "movie\_id", "title", "genres", "user\_id", "gender", "age", "occupation", "zip\_code", "rating" } |

# Hand-craft a small set of documents (5-10) using your new schema which examplify MovieLens data, put them into a new collection, and redesign the above queries so they operate on the new schema.

// Attempt three – using mapReduce merge

var mapFunctionUsers = function() {

var data = {

user\_id: this.\_id,

occupation: this.occupation

}

emit(this.\_id,data);

};

var reduceFunctionUsers = function(key, values) {

var result = {

user\_id: 0,

occupation: 0,

ratings: []

};

values.forEach(function(value) {

var field;

if ("rating" in value) {

if (!("ratings" in result)) {

result.ratings = [];

}

result.ratings.push(value.rating);

}

else if ("occupation" in value) {

result.occupation = value.occupation;

}

result.user\_id = value.user\_id;

});

result.user\_id = key;

return result;

};

db.users.mapReduce(

mapFunctionUsers,

reduceFunctionUsers,

{ out: { reduce: "user\_rating" } }

)

var mapFunctionRatings = function() {

var data = {

user\_id: this.user\_id,

rating: this.rating

}

emit(this.user\_id,data);

};

db.users.mapReduce(

mapFunctionRatings,

reduceFunctionUsers,

{ out: { reduce: "user\_rating2" } }

)

db.users.find().forEach( function(x){db.users2.insert(x)} );

var mapFunctionRatings = function() {

emit(this.user\_id,this.rating);

};

var reduceFunctionRatings = function(key, values) {

var ratings = { };

for (var k = 1; k < values.length; ++k) {

ratings[k] = values[k];

}

return ratings;

};

db.ratings.mapReduce(

mapFunctionRatings,

reduceFunctionRatings,

{ out: { merge: "users2" } }

)

# Optional: Design a (set of) map-reduce functions that will translate the three MovieLens collections into a single collection with your newly defined schema (denormalization). (Tricky, I haven't been able to do it (yet) :)

# Optional: Use two/more machines to explore the auto-sharding facilities in MongoDB.

# Optional: Use three machines to explore the replication facilities in MongoDB.

# Optional: Read the MovieLens data into a SQL database of your choice and compare performance between SELECTS and Map-Reduce. (Postings on the web generally state that Map Reduce is slow in Mongo.)