Indholdsfortegnelse

[Introduction 2](#_Toc343203584)

[Find the ID of the film "The Rock" from 1996? (Use find()): 2](#_Toc343203585)

[Find how many 5-star ratings "The Rock" has received received? (Use find()) 2](#_Toc343203586)

[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](#_Toc343203587)

[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](#_Toc343203588)

[Count how many movies are classifed as 'Animation' movies? 4](#_Toc343203589)

[Generate a collection "drama" containing all titles of movies classified as genre Drama. 5](#_Toc343203590)

[How many users have profession 'writer' (occupation=20)? 5](#_Toc343203591)

[What is the distribution of ratings for The Rock given by writers? (Tricky :) 5](#_Toc343203592)

[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. 8](#_Toc343203593)

[If the genre-query was important we could unfold it further to have one entry per genre (more duplicate data). It is possible to remove some of the original linking data (movie\_id and rating id), but the primary key (\_id) is good to have for deleting/updating and the movie\_id is preferred to a string to check for inconsistencies (someone entering “The Rock” as a title and others using “Rock, The”), however this design do not include a description as to how the movie\_id is generated (or user\_id or occupation for that matter). This is naturally a limitation in the design documentation, but will be left up to the reader to ponder. 9](#_Toc343203594)

[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. 9](#_Toc343203595)

[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) :) 11](#_Toc343203596)

[Optional: Use two/more machines to explore the auto-sharding facilities in MongoDB. 12](#_Toc343203597)

[Optional: Use three machines to explore the replication facilities in MongoDB. 12](#_Toc343203598)

[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.) 12](#_Toc343203599)

# Introduction

This document contains the answer to assignment H6 in the course 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.

Simple using for:

|  |
| --- |
| 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] });  } |

Alternatively 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" }, query: { occupation: 20 } }  )  db.ratings.mapReduce(  mapFunctionRatings,  reduceFunction,  { out: { reduce: "user\_rating" }, query: { movie\_id: 733 } }  ) |

This results in a single collection containing the merge of users and ratings, but only the relevant data and only for writers with reviews of The Rock, as shown below.

|  |
| --- |
| { "\_id" : 350, "value" : { "user\_id" : 350, "occupation" : 20, "is\_rating" : 1, "ratings" : [ 0, 0, 0, 1, 0 ] } }  { "\_id" : 356, "value" : { "user\_id" : 356, "occupation" : 20, "is\_rating" : 0, "ratings" : [ 0, 0, 0, 0, 0 ] } }  { "\_id" : 362, "value" : { "user\_id" : 362, "occupation" : 20, "is\_rating" : 0, "ratings" : [ 0, 0, 0, 0, 0 ] } }  { "\_id" : 382, "value" : { "user\_id" : 382, "occupation" : 20, "is\_rating" : 0, "ratings" : [ 0, 0, 0, 0, 0 ] } }  { "\_id" : 406, "value" : { "user\_id" : 406, "occupation" : 20, "is\_rating" : 0, "ratings" : [ 0, 0, 0, 0, 0 ] } }  { "\_id" : 454, "value" : { "user\_id" : 454, "occupation" : 20, "is\_rating" : 0, "ratings" : [ 0, 0, 0, 0, 0 ] } }  has more |

This collection has 1568 entries. As the user\_id is the key we know that there are 1568 writers who rated a movie. It is possible to query this further:

db.user\_rating.find( { 'value.is\_rating': 1, 'value.occupation': 20 } )

|  |
| --- |
| { "\_id" : 1680, "value" : { "user\_id" : 1680, "occupation" : 20, "is\_rating" : 1, "ratings" : [ 0, 0, 0, 1, 0 ] } }  { "\_id" : 1737, "value" : { "user\_id" : 1737, "occupation" : 20, "is\_rating" : 1, "ratings" : [ 0, 0, 0, 0, 1 ] } }  { "\_id" : 1820, "value" : { "user\_id" : 1820, "occupation" : 20, "is\_rating" : 1, "ratings" : [ 0, 0, 1, 0, 0 ] } }  { "\_id" : 1884, "value" : { "user\_id" : 1884, "occupation" : 20, "is\_rating" : 1, "ratings" : [ 0, 0, 1, 0, 0 ] } }  { "\_id" : 2041, "value" : { "user\_id" : 2041, "occupation" : 20, "is\_rating" : 1, "ratings" : [ 0, 0, 0, 1, 0 ] } }  has more |

A count on this shows that there are 53 writers who rated The Rock. This is true because the is\_rating is only set to 1 if there is a rating for the given user and The Rock and the occupation is only set to 20 if the users occupation is a writer. To extract the distribution of these 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.is\_rating': 1, 'value.occupation': 20 } }  ) |

Success! The distribution if ratings by writers of The Rock is:

|  |
| --- |
| {  "\_id" : 20,  "value" : {  "ratings" : [  4,  7,  15,  18,  9  ]  } |

Or:

|  |  |
| --- | --- |
| Rating 1 | 4 |
| Rating 2 | 7 |
| Rating 3 | 15 |
| Rating 4 | 18 |
| Rating 5 | 9 |

# 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.

In Mongo naturally there are not schemas ☺, but when all is said and done – there is always a schema. However, designing the perfect schema is extremely complicated and is a matter of weighing the collection size against the query speed. It is also a matter of considering what type of queries will be made, how often will they be made and how fast must they respond. Another consideration, which is often overlooked is: how is this going to be deployed. Mongo support distributed deployment, but there is a different in the optimum design if we are talking 1000 machines with a slow internal connection or 2 machines with a extremely fast connection. It all factors into the design. Now, we do not have sufficient information about intended use or deployment, nor do we have sufficient time (there would also have to be time for the authors to learn how to perform such an analysis – maybe after the coming fagpakke from Aalborg☺). For these reasons we will make certain assumptions.

1. Performance is more important than size.
2. High degree of distribution (many distributed servers)
3. Queries on genre is less common and may take longer.

The third assumption allows us to keep the genres in an array. It is possible to query them, but they are not as easily indexed. The existing collections look as follow

|  |
| --- |
| Ratings: { "\_id", "movie\_id", "user\_id", "rating" }  Movies: { "\_id", "title", "genres" }  Users: { "\_id", "gender", "age", "occupation", "zip\_code" } |

Combining these to a single collection:

|  |
| --- |
| { "\_id", "movie\_id", "title", "genres", "user\_id", "gender", "age", "occupation", "zip\_code", "rating", "timestamp" } |

# If the genre-query was important we could unfold it further to have one entry per genre (more duplicate data). It is possible to remove some of the original linking data (movie\_id and rating id), but the primary key (\_id) is good to have for deleting/updating and the movie\_id is preferred to a string to check for inconsistencies (someone entering “The Rock” as a title and others using “Rock, The”), however this design do not include a description as to how the movie\_id is generated (or user\_id or occupation for that matter). This is naturally a limitation in the design documentation, but will be left up to the reader to ponder.

# 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.

|  |
| --- |
| var writeToCollection = function(data) {  for (var i = 0; i < data.length; ++i) {  db.new\_schema.insert(data[i]);  }  };  var x = [ {  \_id: ObjectId("50b5db491d41c80f92000001"),  movie\_id: 1,  title: "Toy Story (1995)",  genres: [ "Animation", "Children's", "Comedy" ],  user\_id: 2,  gender: "F",  age: 56,  occupation: 16,  zip\_code: 70072,  rating: 5,  timestamp: "978302039"  },{  \_id: ObjectId("50b5db491d41c80f92000002"),  movie\_id: 2,  title: "Jumanji (1995)",  genres: [ "Adventure", "Children's", "Fantasy" ],  user\_id: 2,  gender: "F",  age: 56,  occupation: 16,  zip\_code: 70072,  rating: 3,  timestamp: "978302109"  },{  \_id: ObjectId("50b5db491d41c80f92000003"),  movie\_id: 3,  title: "Grumpier Old Men (1995)",  genres: [ "Comedy", "Romance" ],  user\_id: 2,  gender: "F",  age: 56,  occupation: 16,  zip\_code: 70072,  rating: 3,  timestamp: "978301968"  },{  \_id: ObjectId("50b5db491d41c80f92000004"),  movie\_id: 2,  title: "Jumanji (1995)",  genres: [ "Adventure", "Children's", "Fantasy" ],  user\_id: 3,  gender: "M",  age: 25,  occupation: 15,  zip\_code: 55117,  rating: 4,  timestamp: " 978300275"  },{  \_id: ObjectId("50b5db491d41c80f92000005"),  movie\_id: 2,  title: "Jumanji (1995)",  genres: [ "Adventure", "Children's", "Fantasy" ],  user\_id: 4,  gender: "M",  age: 45,  occupation: 7,  zip\_code: 02460,  rating: 5,  timestamp: " 978824291"  },{  \_id: ObjectId("50b5db491d41c80f92000006"),  movie\_id: 1,  title: "Toy Story (1995)",  genres: [ "Animation", "Children's", "Comedy" ],  user\_id: 4,  gender: "M",  age: 45,  occupation: 7,  zip\_code: 02460,  rating: 3,  timestamp: " 978302268"  },{  \_id: ObjectId("50b5db491d41c80f92000007"),  movie\_id: 1,  title: "Toy Story (1995)",  genres: [ "Animation", "Children's", "Comedy" ],  user\_id: 3,  gender: "M",  age: 25,  occupation: 15,  zip\_code: 55117,  rating: 5,  timestamp: " 978302039"  },{  \_id: ObjectId("50b5db491d41c80f92000008"),  movie\_id: 733,  title: "Rock, The (1996)",  genres: [ "Action", "Adventure", "Thriller" ],  user\_id: 3,  gender: "M",  age: 25,  occupation: 15,  zip\_code: 55117,  rating: 1,  timestamp: " 978302039"  },{  \_id: ObjectId("50b5db491d41c80f92000009"),  movie\_id: 733,  title: "Rock, The (1996)",  genres: [ "Action", "Adventure", "Thriller" ],  user\_id: 4,  gender: "M",  age: 45,  occupation: 7,  zip\_code: 02460,  rating: 4,  timestamp: " 978302268"  },{  \_id: ObjectId("50b5db491d41c80f9200000a"),  movie\_id: 733,  title: "Rock, The (1996)",  genres: [ "Action", "Adventure", "Thriller" ],  user\_id: 2,  gender: "F",  age: 56,  occupation: 16,  zip\_code: 70072,  rating: 5,  timestamp: "978301968"  } ];  writeToCollection(x); |

With this we can rewrite the queries from above.

1.

|  |
| --- |
| db.new\_schema.findOne( { title: " Rock, The (1996)" },{ \_id: 0, movie\_id: 1 }); |

2.

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

3.

|  |
| --- |
| Identical to the previous |

4.

|  |
| --- |
| Same as previous |

5.

|  |
| --- |
| Same |

6.

|  |
| --- |
| Same |

7.

|  |
| --- |
| db.new\_schema.distinct("user\_id", { occupation: 20 }).length; |

8.

|  |
| --- |
| var mapFunction1 = function() {  emit(this.rating, 1);  };  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.is\_rating': 1, 'value.occupation': 20 } }  ) |

# 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) :)

We are going to attempt the following procedure:

1. Use mapReduce to merge the ratings with the users based on user id. As multiple ratings exist for each movie there will be an array of movies and ratings for each user.
2. To unfold the result we have not found an elegant solution, so a simple loop creating the new collection is used.
3. Use mapReduce to merge the new collection with movies based on movie id. As multiple ratings exist for each movie there will be an array of ratings for each movie.
4. Finally we unfold the result into the final merged collection.

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

Unfortunately this does not work. Either the user info is overridden or the ratings array is.

# 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.)