SMD: Application Proposal & Database Implementation
DATA 225 2022, GROUP 4

Razan Dababo, Priya Khandelwal, and Maria Poulose
San Jose State University, razan.dababo, priya.khandelwal, mariapoulose@sjsu.edu

Abstract **- In this report, we propose and implement a movie database using a NoSQL database management system (DBMS) design executed in MongoDB that would form the core of our movie-streaming application, SMD. We provide the schema model of our database, describe user functionalities, provide user-specific queries, implement triggers, manage access privileges, and evaluate the performance of our database.**

Index Terms - Denormalize, MongoDB, NoSQL

INTRODUCTION

Our proposed application, San Jose State University Movie Database (SMD), is a subscription-based streaming service that allows our members to watch movies without commercials on an internet-connected device. Our application will provide users a hassle-free experience to select from a large number of films based on various search criteria, such as rating, actor name, or genre, and access their history of watched movies. Our application will also allow managers and system administrators to grant access privileges to various users of the application and track information such as subscription details and customer service issues.

SOLUTION REQUIREMENTS

Given the large volume and interrelated nature of the data to be handled by our SMD application, our first instinct was to store and organize such data using a relational database management system (RDBMS).

However, we realized that RDBMS design has several shortcomings. First, RDBMSs do not work well with unstructured or semi-structured data due to schema and type constraints, and are hence ill-suited for large analytics. Second, schema constraints in RDBMS require that schemas and types be generally identical between source and destination tables when migrating one RDBMS to another. Third, the tables in a relational database may not necessarily map one-to-one with an object or class representing the same data. And finally, complex datasets or datasets containing variable-length records are generally difficult to handle with an RDBMS schema.

Thus, we decided to employ a NoSQL DBMS using MongoDB, which is a source-available cross-platform document-oriented database program. The advantages of using NoSQL DBMS are several. First, NoSQL DBMS is schema-free, which would allow the use of both unstructured and semi-structured data. Second, NoSQL is highly scalable (horizontally) through sharding. Third, NoSQL has cloud and big data support. And finally, they boast high speed and require relatively little database administration in comparison to RDBMS.

Our application has the following limitations:

- The application will not have a movie-recommendation functionality.
- Users will not be able to fetch movies based on latest additions or movie language.
- Users cannot rate movies.

DENORMALIZATION

The challenge we faced in using NoSQL DBMS was denormalizing our previous RDBMS design. Figure 1 shows the original Entity Relationship Diagram of SMD. Tables *movie*, *Movie_cast*, *Actor*, *Movie_genres*, *Genres* were denormalized into a single collection labeled *movies* in our MongoDB database.

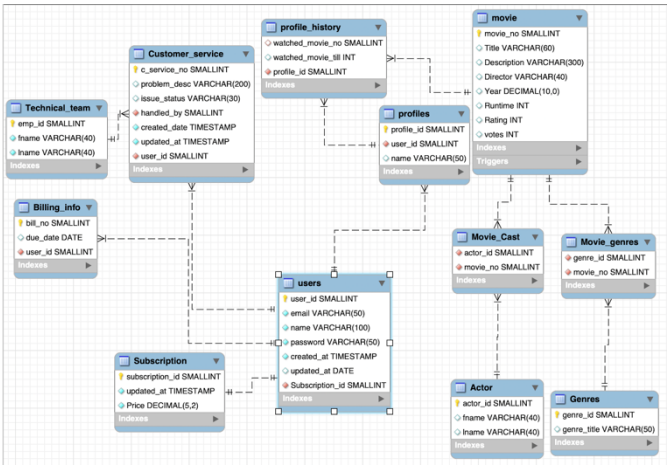

FIGURE 1. The original RDBMS Entity Relationship (ER) diagram above details the attributes, participation, and cardinalities among the 12 tables that contain our data.

Tables *users*, *Subscription*, *Billing_info*, *profiles*, *profile_history*, *Customer_service*, and *Technical_team* were denormalized into a single collection labeled *users* in MongoDB.

The *movies* collection consists of 1,000 documents, each of which represents a movie record (or row in previous RDBMS *movie*-related tables), as seen in Figure 2. To avoid over-redundancy and facilitate accessibility, the *genres* and *actors* fields were stored in arrays consisting of each movie's list of genres and actors, respectively.

The *users* collection consists of 300 documents, each of which represents a customer record (or row in previous RDBMS *users*-related tables), as seen in Figure 2. For the same reasons listed above, the *billing_info*, *subscription, customer_service,* and *profiles* fields were stored in arrays consisting of each customer's list of subscription, billing, service complaint, and linked profiles information.

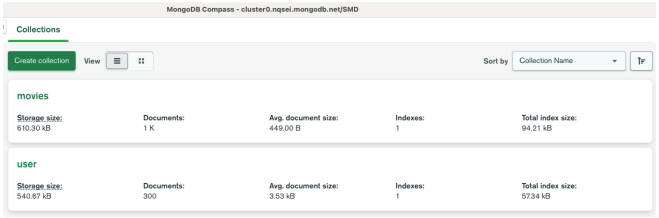

FIGURE 2. Two collections make up our SMD NoSQL database: movies and user. Each employs a single default index.

CONCEPTUAL DATABASE DESIGN & SCHEMA

Since our SMD application serves a dual purpose – it serves as both a movie database as well as a means for managers to maintain the application and track customer service issues and movies watched among other things – the challenge of designing a NoSQL MongoDB schema was denormalizing our original SQL schema such that no data is sacrificed and the schema is not too complex. Our original plan was to denormalize our SQL schema into a single NoSQL collection. However, we realized that design was not optimal since not all movies will necessarily have a watch history; having one collection would result in too many lengthy, embedded documents.

Hence, we decided to split our SQL schema into two collections: movies collection and user collection, as seen in Figures 3 and 4. The two collections are joined on _id in movies collection and watched_movie_no within the history array of the profiles array within user collection (see Figure 5); in other words, the watched_movie_no (of character type Int32) in the user collection, which documents what movies each profile has watched, points to the custom object id (of character Int 32) within the movies collection. In this way, we were able to avoid an overly complex, single collection and instead create a more logical and efficient schema structure within MongoDB.

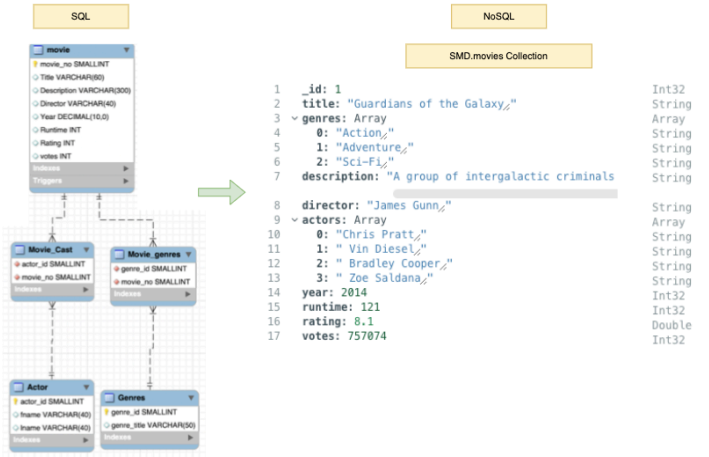

FIGURE 3. All movie-related SQL tables were denormalized into a single SMD.movies collection. The document structure of SMD.movies collection can be seen on the right side of the figure.

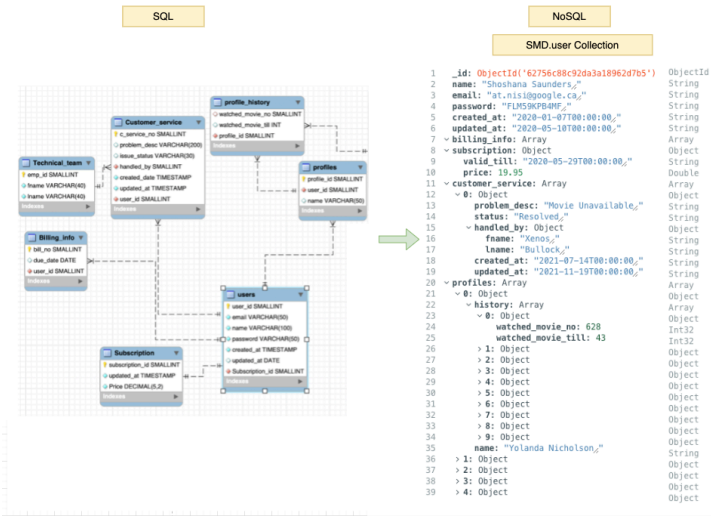

FIGURE 4. All user-related SQL tables were denormalized into a single SMD.user collection. The document structure of SMD.user collection can be seen on the right side of the figure.

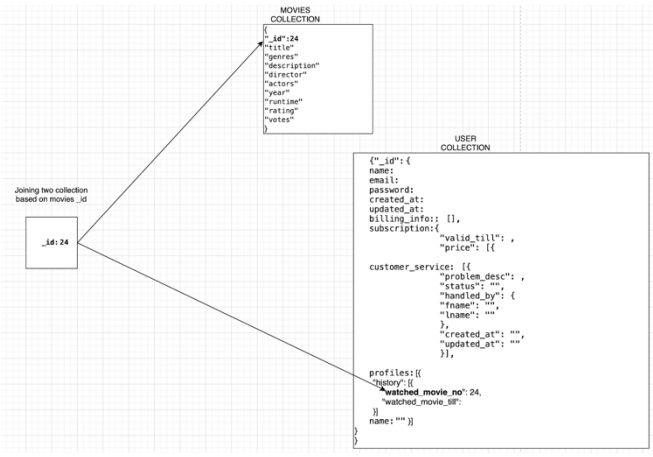

FIGURE 5. The two collections are joined by profiles.history.watched_movie_no in user collection, which points to _id in movie collection.

FUNCTIONAL ANALYSIS

I. Defining Database User Types

Our application will cater to three types of users, as outlined below:

- Customers: users who will access the database to find information about movies. This group would search the database for common information about the movies, such as their actors or genres.
- Managers: users who will have unrestricted access to all tables and insert data into, update, or delete data from the database as needed.
- Administrators: users who will manage access privileges and upgrade/alter/program the database as needed.

II. Functional Components

Each customer account in SMD can have up to 5 profiles, each profile having access to query and watch movies. Each account will automatically be billed a fixed-amount fee of $19.95 monthly. Users can report issues related to their account, and customer service issues are handled by the technical team, who have access to all customer information, including watch history, subscription, billing information, etc. The functional component of each query and aggregation will be discussed in further detail in the Queries section.

QUERIES

Below, we include examples of the queries/aggregations that the users of the SMD application might find useful, along with query code, outputs, and database interactions.

I. Customer Queries (CU)

The following queries are specifically designed from a customer's perspective:

CU1) This query allows customer (e.g. email = 'ornare@aol.edu') to check how many movies were watched by a particular profile within their account (e.g. profile.name = 'Ira Clark') and how much of the movie they watched (watched until).

```
db.user.aggregate([
   {
      '$lookup': {
         'from': 'MOVIE',
         'localField': 'profiles.history.watched_movie_no',
         'foreignField': '_id',
         'as': 'Movie'
      }
   }, {'$unwind': '$profiles'
}, {
      '$match': {'$and':[
{'email': 'ornare@aol.edu'},
{'profiles.name': 'Ira Clark'}
      ]
   }
   }, {
      '$project': {
         'profiles.history.watched_movie_no': 1,
         'profiles.history.watched_movie_till': 1
      }
   }
]).pretty()
```

```
> db.user.aggregate([
    {
        '$lookup': {
            'from': 'MOVIE',
            'localField': 'profiles.history.watched_movie_no',
            'foreignField': '_id',
            'as': 'Movie'
        }
    }, {'$unwind': '$profiles'
}, {
        '$match': {'$and':[
{'email': 'ornare@aol.edu'},
{'profiles.name': 'Ira Clark'}
        ]
     }
    }, {
        '$project': {
            'profiles.history.watched_movie_no': 1,
            'profiles.history.watched_movie_till': 1
        }
    }
]).pretty()
< { _id: ObjectId("62756c88c92da3a18962d7f2"),
  profiles:
   { history:
      [ { watched_movie_no: 505, watched_movie_till: 2 },
        { watched_movie_no: 354, watched_movie_till: 6 },
        { watched_movie_no: 94, watched_movie_till: 21 },
        { watched_movie_no: 518, watched_movie_till: 48 },
```

FIGURE 6. Input and truncated output of query CU1.

CU2) This query allows customer to fetch movie(s) based on a given actor (e.g. 'Chris Pratt'). This query searches within the movies collection for a match to the actor's name in the actor field and displays only the fields which are specified in the query: title, actors, and year.

db.movies.find({ "actors": "Chris Pratt"}, {"actors":1, "title":1, 'year':1, _id:0})

```
> db.movies.find({ "actors": "Chris Pratt"}, {"actors":1,"title":1, 'year':1, _id:0})

< { title: 'Guardians of the Galaxy',
    actors:
     [ 'Chris Pratt',
       ' Vin Diesel',
       ' Bradley Cooper',
       ' Zoe Saldana' ],
    year: 2014 }
  { title: 'Jurassic World',
    actors:
     [ 'Chris Pratt',
       ' Bryce Dallas Howard',
       ' Ty Simpkins',
       'Judy Greer' ],
    year: 2015 }
  { title: 'The Lego Movie',
    actors:
     [ 'Chris Pratt',
       ' Will Ferrell',
       ' Elizabeth Banks',
       ' Will Arnett' ],
    year: 2014 }
```

FIGURE 7. **Input and output of query CU2.**

CU3) This query allows customer to fetch movie(s) based on a given genre (e.g. 'Horror'). This query searches within the movies collection for a match to the given genre in the genre field and displays only the fields which are specified in the query: title, genres, and year.

db.movies.find({ "genres": "Horror"}, {"title":1, "genres":1, 'year':1, _id:0})

```
> db.movies.find({ "genres": "Horror"}, {"title":1, "genres":1, 'year':1, _id:0})
< { title: 'Split', genres: [ 'Horror', 'Thriller' ], year: 2016 }
  { title: 'Hounds of Love',
    genres: [ 'Crime', 'Drama', 'Horror' ],
    year: 2016 }
  { title: 'Dead Awake',
    genres: [ 'Horror', 'Thriller' ],
    year: 2016 }
  { title: 'Resident Evil: The Final Chapter',
    genres: [ 'Action', 'Horror', 'Sci-Fi' ],
    year: 2016 }
  { title: 'Don\'t Fuck in the Woods',
    genres: [ 'Horror' ],
    year: 2016 }
  { title: 'Don\'t Breathe',
    genres: [ 'Crime', 'Horror', 'Thriller' ],
    year: 2016 }
  { title: 'The Autopsy of Jane Doe',
    genres: [ 'Horror', 'Mystery', 'Thriller' ],
    year: 2016 }
  { title: 'The Void',
    genres: [ 'Horror', 'Mystery', 'Sci-Fi' ],
    year: 2016 }
  { title: 'The Belko Experiment',
```

FIGURE 8. **Input and truncated output of query CU3.**

CU4) This query allows customer to fetch details of profiles linked to their account. This query searched in the user collection for a match to the given email in the email field, then displays only the fields specified in the query: account email and corresponding profile names.

db.user.find({ "email": "at.nisi@google.ca"}, {"email":1, "profiles.name":1,_id:0})

```
> db.user.find({ "email": "at.nisi@google.ca"}, {"email":1, "profiles.name":1,_id:0})
< { email: 'at.nisi@google.ca',
    profiles:
     [ { name: 'Yolanda Nicholson' },
       { name: 'Amir Pace' },
       { name: 'Ebony Pickett' },
       { name: 'Murphy Hardin' },
       { name: 'Knox Clark' } ] }
Atlas atlas-w4c6nx-shard-0 [primary] SMD>
```

FIGURE 9. **Input and output of query CU3.**

I. Manager Queries (MA)

These queries are specifically designed from the manager's perspective:

MA1) This query allows manager to fetch customers whose bill is due on the present day.

```
db.user.aggregate([
   {
     '$match': {"billing_info.due_date":"2022-05-07T00:00:00"}
     },
     {
     '$project': {
        'name': 1,
        'email': 1
     }
   }
])
```

```
> db.USER.aggregate([
     {
         '$match': {"billing_info.due_date":"2022-05-07T00:00:00"}
         },
         {
         '$project': {
             'name': 1,
             'email': 1
         }
     }
  ]).pretty()
< { _id: ObjectId("626f3cfc572a5f66857dd78f"),
    name: 'Debra Bailey',
    email: 'risus.odio@aol.com' }
  { _id: ObjectId("626f3cfc572a5f66857dd87a"),
    name: 'Barbara Saunders',
    email: 'lacus.quisque@outlook.net' }
  { _id: ObjectId("626f3cfc572a5f66857dd8a7"),
    name: 'Leilani Taylor',
    email: 'rhoncus.id.mollis@hotmail.net' }
  { _id: ObjectId("626f3cfc572a5f66857dd8b2"),
    name: 'Melodie Patrick',
    email: 'pede@outlook.com' }
Atlas atlas-hz89k7-shard-0 [primary] SMD>
```

FIGURE 10. **Input and truncated output of query MA1.**

MA2) This aggregation allows manager to fetch details of monthly subscription numbers, which helps them to get the count of new SMD customers in a given month (e.g. February, 2020). This aggregation looks for a match to the specified date range within the created_at field, then counts the number of resulting values.

```
db.user.aggregate([{
   $project: { _id:0,
      'created_at': 1
   }
}, {
   $match: {
      'created_at': {
         $gte: '01-02-20',
         $lte: '28-02-20'
      }
   }
},
{
   $count: 'Number of new Customers in February'
}])
```

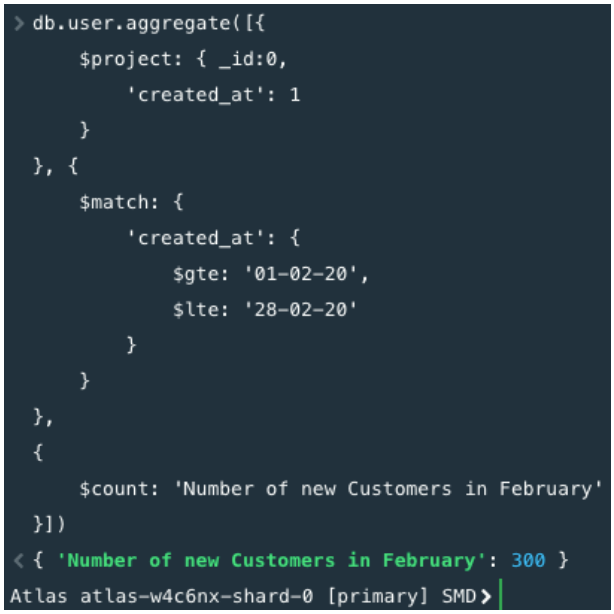

FIGURE 11. **Input and output of query MA2.**

MA3) This aggregation allows manager to fetch details of customer service issue status, which helps them track how many customer service problems are in active/resolved status. This aggregation first projects the desired fields from the *user* collection: customer service status and customer service created date. The aggregation then sorts the results in descending order of created date and ascending alphabetical order of status.

```
db.user.aggregate([{$project: {
 'customer_service.status': 1,'customer_service.created_at': 1,
 _id: 0
}},
{$sort: {
 'customer_service.created_at': -1, 'customer_service.status': 1}
}])
```

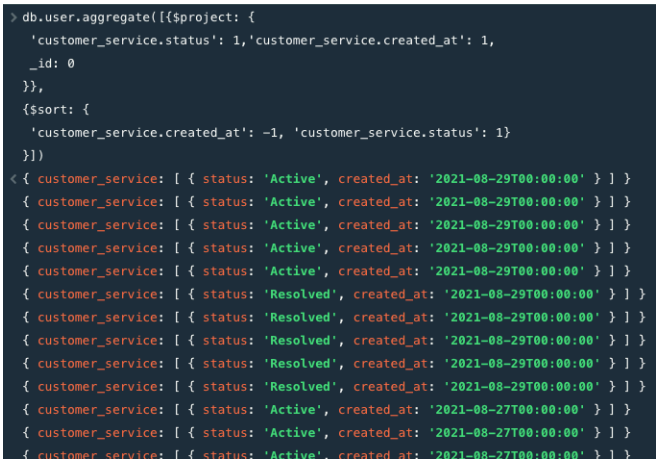

FIGURE 12. **Input and truncated output of query MA3.**

MA4) This aggregation allows manager to fetch total number of problems faced by users in a month (e.g. July, 2021). This aggregation first projects the desired fields from the user collection, then looks for a match for the specified date range within the customer service *created_at* field, then prints total count of results in the new field.

```
db.user.aggregate([{$project: {
 'customer_service.problem_desc': 1,
 'customer_service.created_at': 1,
 _id: 0
}}, {$match: {
 'customer_service.created_at': {
  $gte: '2021-07-01',
  $lt: '2021-07-31'
 }
}}, {$count: 'Total Number of Customer Service Issues in July 2021'}])
```

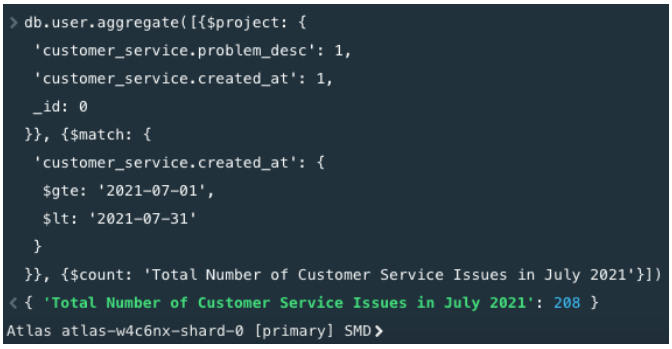

FIGURE 13. **Input and output of query MA4.**

MA5) This aggregation allows manager to fetch list of which employees resolved which customer service issue. This aggregation first projects the desired fields from the user collection, then unwinds the array that has been projected, then looks for a match to the specified status, then sorts the results by ascending order of created date, employee last name, and name of customer service issue.

```
db.user.aggregate([{$project: {
 customer_service: 1,
 _id: 0
}}, {$unwind: {
 path: '$customer_service'
}}, {$match: {
 'customer_service.status': 'Resolved'
}}, {$sort: {
 'customer_service.created_at': 1,
 'customer_service.handled_by.lname': 1,
 'customer_service.problem_desc': 1
}}])
```

FIGURE 14. **Input and truncated output of query MA5.**

MA6) This aggregation allows manager to fetch customer service issue counts by category for a given month (e.g. July, 2021). The aggregation begins by projecting the desired fields from user collection and unwinding the resulting array, then looking for a match to the specified date range within the customer service created date field, then grouping the results by customer service issue, counting the values in each group, then sorting the results by descending order of counts.

```
db.user.aggregate([{$project: {
 customer_service: 1,
 _id: 0
}}, {$unwind: {
 path: '$customer_service'
}}, {$match: {
 'customer_service.created_at': {
  $gte: '2021-07-01',
  $lt: '2021-07-31'
 }
}}, {$group: {
 _id: '$customer_service.problem_desc',
 NumberComplaintsJuly2021: {
  $count: {}
 }
}}, {$sort: {
 NumberComplaintsJuly2021: -1
}}])
```

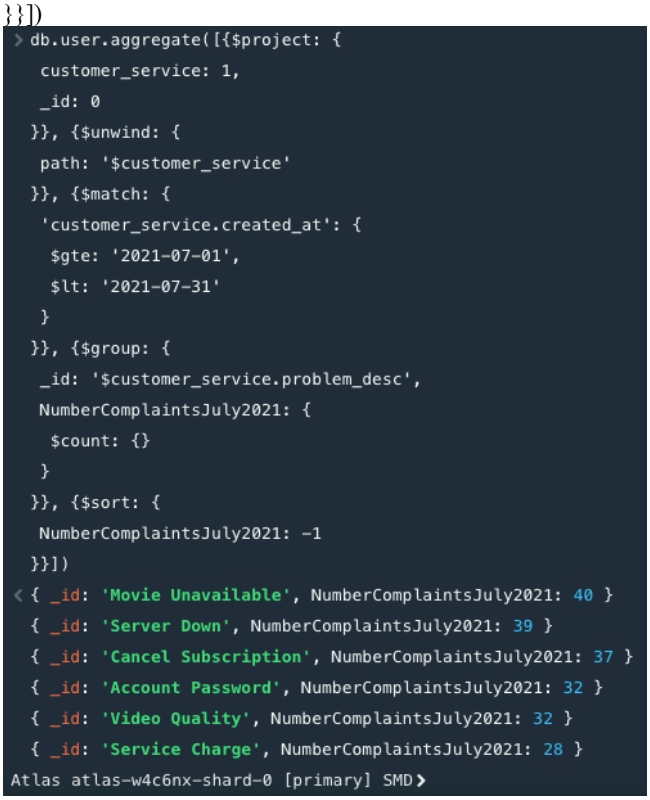

FIGURE 15. **Input and output of query MA6.**

MA7) This aggregation allows manager to fetch which employee handled the most customer service issues in a particular month (for example, July, 2021). The aggregation begins by projecting the desired fields from user collection and unwinding the resulting array, then looking for a match to the specified date range

within the customer service created date field, then grouping the results by employee last name, counting the values in each group, then sorting the results by descending order of counts.

```
db.user.aggregate([{$project: {
 customer_service: 1
}}, {$unwind: {
 path: '$customer_service'
}}, {$match: {
 'customer_service.created_at': {
  $gte: '2021-07-01',
  $lt: '2021-07-31'
 }
}}, {$group: {
 _id: '$customer_service.handled_by.fname',
 TotalComplaintsHandled: {
  $count: {}
 }
}}, {$sort: {
 TotalComplaintsHandled: -1
}}])
```

```
> db.user.aggregate([{$project: {
    customer_service: 1
  }}, {$unwind: {
    path: '$customer_service'
  }}, {$match: {
    'customer_service.created_at': {
      $gte: '2021-07-01',
      $lt: '2021-07-31'
    }
  }}, {$group: {
    _id: '$customer_service.handled_by.lname',
    TotalComplaintsHandled: {
      $count: {}
    }
  }}, {$sort: {
    TotalComplaintsHandled: -1
  }}])
< { _id: 'Joseph', TotalComplaintsHandled: 15 }
  { _id: 'Johnston', TotalComplaintsHandled: 15 }
  { _id: 'Cote', TotalComplaintsHandled: 14 }
  { _id: 'Mendez', TotalComplaintsHandled: 14 }
  { _id: 'Luna', TotalComplaintsHandled: 14 }
  { _id: 'Moses', TotalComplaintsHandled: 13 }
  { _id: 'Craig', TotalComplaintsHandled: 13 }
  { _id: 'Bullock', TotalComplaintsHandled: 13 }
  { _id: 'Pratt', TotalComplaintsHandled: 11 }
  { _id: 'Fitzgerald', TotalComplaintsHandled: 11 }
  { _id: 'Ramsey', TotalComplaintsHandled: 11 }
  { _id: 'Andrews', TotalComplaintsHandled: 10 }
  { _id: 'Peterson', TotalComplaintsHandled: 10 }
  { _id: 'Terry', TotalComplaintsHandled: 9 }
  { _id: 'Aguirre', TotalComplaintsHandled: 8 }
  { _id: 'Strickland', TotalComplaintsHandled: 7 }
  { _id: 'Conrad', TotalComplaintsHandled: 7 }
```

FIGURE 16. Input and truncated output of query MA7.

MA8) This aggregation allows manager to fetch number of impacted customers in a given month (e.g. July, 2021). The aggregation begins by projecting the desired fields from user collection and unwinding the resulting array, then looking for a match to the specified date range within the customer service created date field, then grouping the results by account email, then counting the total results and printing the results to a new field.

```
db.user.aggregate([{
   $project: {
      email: 1,
      'customer_service.created_at': 1
   }
}, {
   $match: {
      'customer_service.created_at': {
         $gte: '2021-01-01',
         $lte: '2021-07-31'
      }
   }
}, {
   $group: {
      _id: '$email'
   }
}, {
   $count: 'Number Impacted Customers in July'
}])
```

```
> db.user.aggregate([{
      $project: {
          email: 1,
          'customer_service.created_at': 1
      }
  }, {
      $match: {
          'customer_service.created_at': {
              $gte: '2021-01-01',
              $lte: '2021-07-31'
          }
      }
  }, {
      $group: {
          _id: '$email'
      }
  }, {
      $count: 'Number Impacted Customers in July'
  }])
< { 'Number Impacted Customers in July': 208 }
Atlas atlas-w4c6nx-shard-0 [primary] SMD>
```

FIGURE 17. Input and output of query MA8.

MA9) This aggregation allows manager to rank all archived customer service issues by number, sorted highest to lowest. This helps manager identify the most frequent customer service issues to-date. The aggregation begins by projecting the desired fields from user collection and unwinding the resulting array, then grouping the results by customer service issue, counting the values in each group, then sorting the results by descending order of counts.

```
db.user.aggregate([{
   $project: {
      customer_service: 1,
      _id: 0
   }
}, {
   $unwind: {
      path: '$customer_service'
   }
}, {
   $group: {
      _id: '$customer_service.problem_desc',
      NumberComplaints: {
         $count: {}
      }
   }
}, {
   $sort: {
      NumberComplaints: -1
   }
}])
```

FIGURE 18. Input and truncated output of query MA9.

MA10) This aggregation allows manager to identify the top 5 watched genres in SMD. The aggregation begins by joining both user and movies collections using $lookup, then unwinds both the movie and genres arrays. It then groups by genres, counts the values in each group, and sorts the results by descending order of counts.

```
db.user.aggregate([{$lookup: {
 from: 'movies',
 localField: 'profiles.history.watched_movie_no',
 foreignField: '_id',
 as: 'Movie'
}}, {$unwind: {
 path: '$Movie'
}}, {$unwind: {
 path: '$Movie.genres'
}}, {$group: {
 _id: '$Movie.genres',
 count: {
  $count: {}
 }
}}, {$sort: {
 count: -1
}}, {$limit: 5}]).pretty()
```

FIGURE 19. Input and truncated output of query MA9.

CLOUD CONNECTIVITY

To connect to MongoDB, we retrieved the hostname and port information from Cloud Manager and then used a MongoDB client (mongosh) to connect. To connect to cluster, we retrieved the hostname and port for the mongos process. Figures 20 – 22 illustrate our cloud connectivity using cloud.mongodb.com and MongoDB Compass.

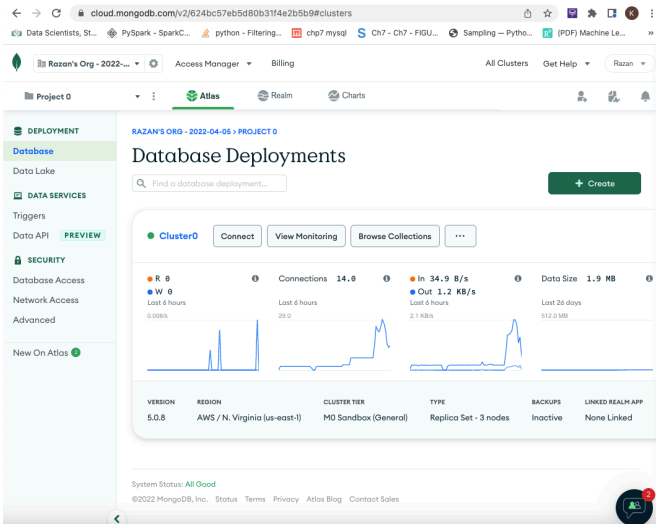

FIGURE 20. This figure shows our database deployments in MongoDB cloud, which includes the cluster (Cluster 0) used for the SMD database.

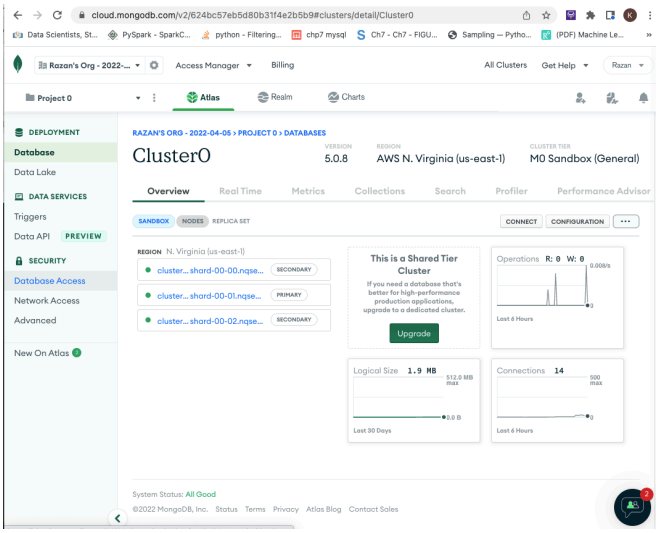

FIGURE 21. This figure gives an overview of Cluster0 in MongoDB cloud, including the hosts, cluster tier, and region.

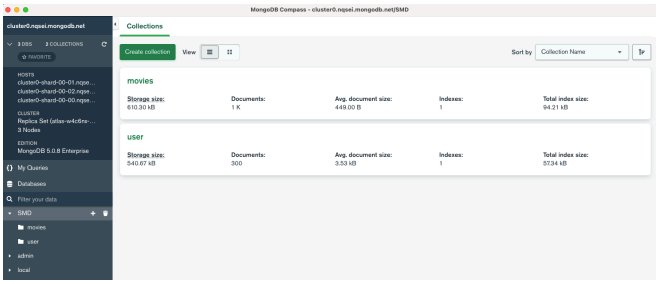

FIGURE 22. This figure shows our connection to the cluster in MongoDB Compass using the hosts.

We have also used PyMongo library to connect MongoDB atlas to Python. PyMongo is a Python distribution containing tools for working with MongoDB, and is the recommended way to work with MongoDB from Python. Included with this report submission is the mongo-connection.ipynb jupyter notebook (Figure 23) which shows the MongoDB Atlas connection and output of the queries showcased in this report.

FIGURE 23. This figure shows the jupyter notebook where a connection to MongoDB Atlas was established.

CHARTS AND VISUALIZATIONS

After connecting to our database from MongoDB cloud, we were able to create meaningful charts and visualizations from our data. Some of these charts are visualizations of the previously discussed queries themselves. Other charts were created on an ad-hoc basis. The charts were saved in our project dashboard on MongoDB.com, as seen in Figure 24.

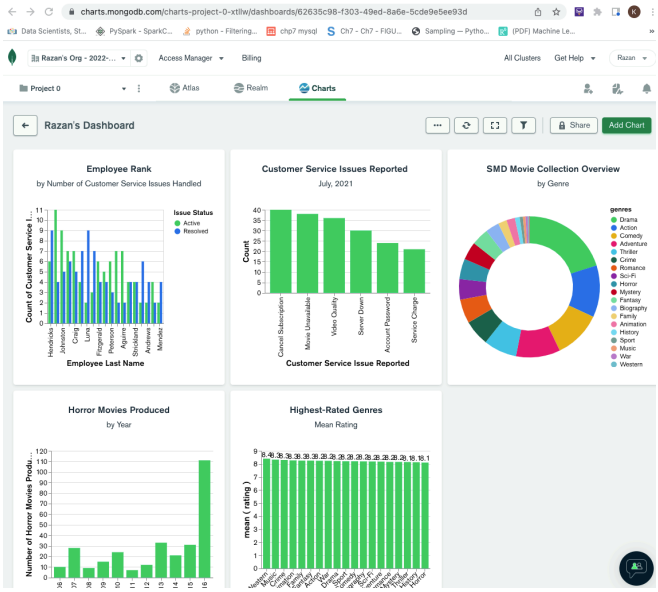

FIGURE 24. Charts Dashboard in MongoDB.com. This dashboard stores saved visualizations and charts, to be used by SMD management.

In Figure 25, we can see the selection of genres available in the SMD movie application, which shows that the largest genre is Drama, followed by Action then Comedy.

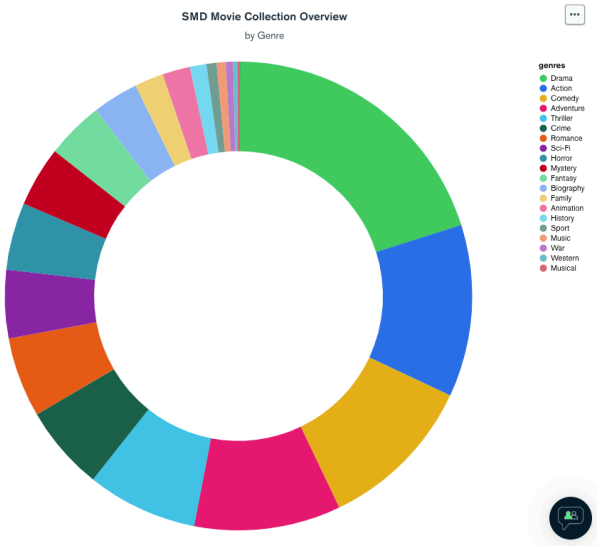

FIGURE 25. **Selection of movie genres available in SMD.**

The chart in Figure 26 was created on an ad-hoc basis by management to get a sense of the trends in the movie industry, while the visual in Figure 27 enhances query MA3 by showing total volume of customer service issues to-date, grouped by status.

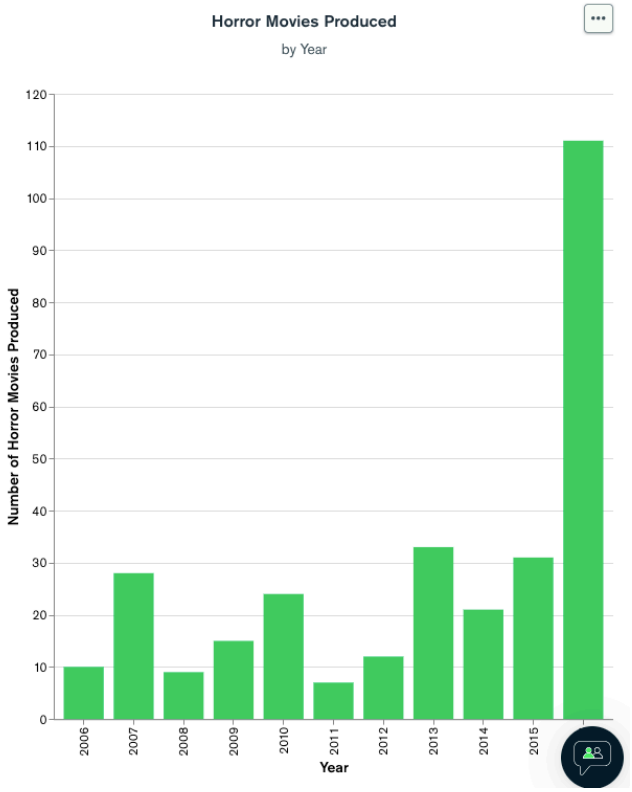

FIGURE 26. **Trends in horror movies produced, 2006 – 2016.**

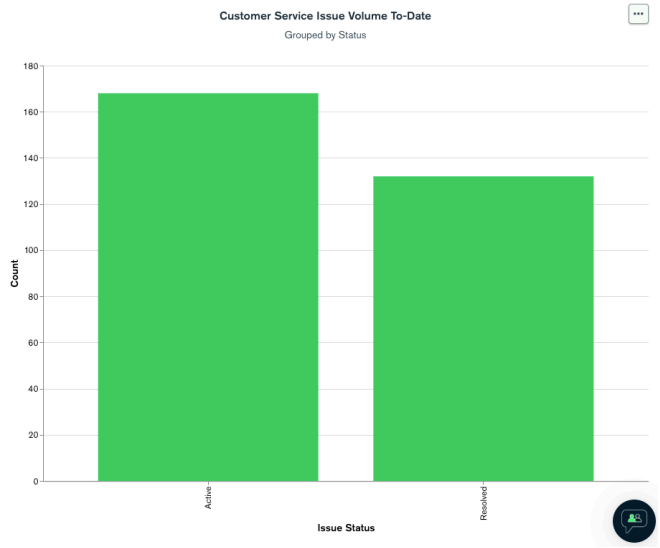

FIGURE 27. **Customer service issue volume to-date, grouped by issue status. This visual is an enhancement of query MA3.**

Chart in Figure 28 was created on an ad-hoc basis by management to get a sense of which genres are highest-rated. The chart shows, surprisingly, Western movies have the highest mean (average) rating.

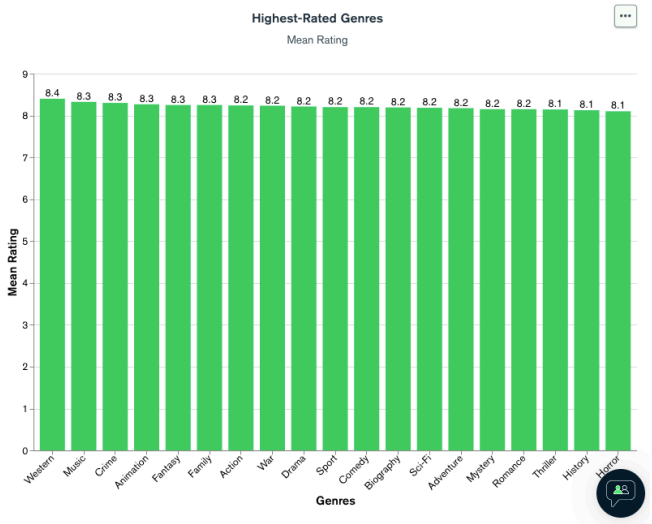

FIGURE 28. **Ad-hoc visual showing highest-rated genres, by mean (average) rating.**

The chart in Figure 29 is a visual representation of query MA6, which displays customer service issues reported in July and groups the counts by category.

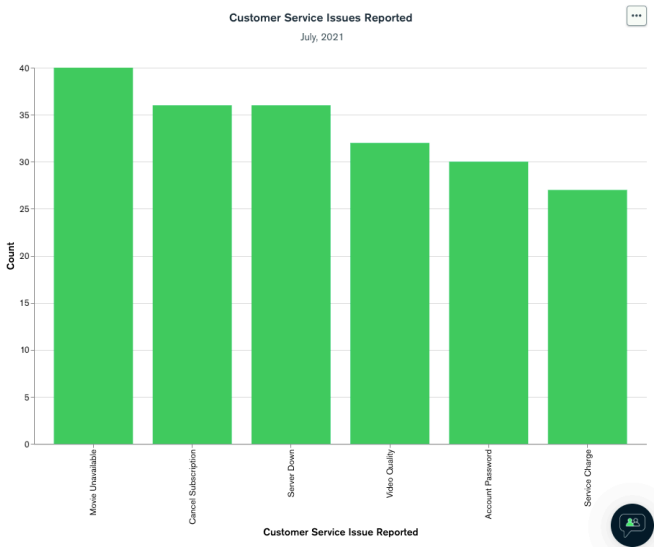

FIGURE 29. **This chart is a visual representation of query MA6.**

The chart in Figure 30 is a visual representation of query MA7, which ranks employees by the number of customer service issues they handled in July, 2021. However, the figure further enhances the query by breaking down each employee's numbers by issue status.

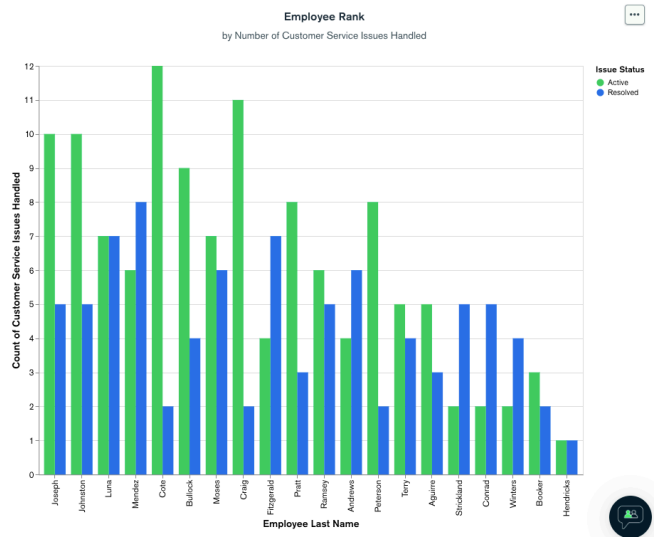

FIGURE 30. **This chart is a visual enhancement of query MA7.**

EVENT SCHEDULER FOR AUTOMATED BILLING

The billing system is automated using a trigger so that monthly statements can be generated for all valid customers. In the SMD billing system, the billing statement is generated every 24 hours for the bills that are due that day and the due date is inserted in the billing_info field in user collection. Below is the trigger code:

```
exports = function() {
  console.log('running..')

   const collection =
context.services.get("Cluster0").db("SMD").collection("USER");
    const doc = collection.updateMany(
        { updated_at: get_prev_DateString() },
        {$set: { updated_at: get_now_DateString()}, $push: {
              billing_info: { due_date: get_now_DateString()}
            }
        },
        );

};

function get_prev_DateString(){
   var date = new Date();
    var month = date.getMonth();
    var year = date.getFullYear();
    var day = date.getDate();
    var m, d;
    if(month>=10){
    m = month;
    }
    else{
    m = `0${month}`;
    }
    if(day>=10){
    d = day;
    }
    else{
    d = `0${day}`;
    }

    str = `${year}-${m}-${d}T00:00:00`;
    console.log('prev', str);
    return str;
}
function get_now_DateString(){
   var date = new Date();
   console.log('date', date)
    var month = date.getMonth()+1;
    var year = date.getFullYear();
    var day = date.getDate();
    var m, d;
    if(month>=10){
    m = month;
    }
    else{
    m = `0${month}`;
    }
    if(day>=10){
    d = day;
    }
    else{
    d = `0${day}`;
    }

    str = `${year}-${m}-${d}T00:00:00`;
    console.log('now', str);
    return str;
}
```

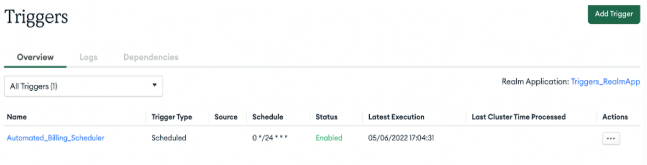

FIGURE 31. **This figure shows the billing trigger created in MongoDB.**

NoSQL Performance

Query performance can be easily measured in MongoDB using the database profiler, which gathers detailed information about database commands executed, including CRUD operations as well as configuration and administration commands. However, this feature of MongoDB is not available to free-tier users. Hence, we have resorted to MongoDB Compass's Explain Plan tab and explain.executionStats method. The Explain Plan tab can be utilized for non-aggregation queries and displays statistics about the performance of a query, including execution time and if/how a query uses an index. The explain.executionStats method can be utilized for both aggregation- and non-aggregation queries and provides statistics that describe the completed query execution, including execution time.

Using the above two methods, we analyze performance of each of the queries and aggregations in the QUERIES section and compare the results to the previously reported performance of MySQL RDBMS. We illustrate the methodology used to conduct the comparison using queries CU2 and MA8.

We begin with query CUS2. As seen in Figure 32, the duration time, or execution time, of this query in SQL was 0.082 seconds, or 82 milliseconds. In contrast, the execution of this query in NoSQL was approximately 2 milliseconds, as seen in Figure 33 and the truncated execution stats below:

```
db.movies.explain("executionStats").find({ "actors":
"Chris Pratt"}, {"actors":1, "title":1, 'year':1, _id:0})
{ explainVersion: '1',
 queryPlanner:
  { namespace: 'SMD.movies',
    indexFilterSet: false,
    parsedQuery: { actors: { '$eq': 'Chris Pratt' } },
    maxIndexedOrSolutionsReached: false,
    maxIndexedAndSolutionsReached: false,
    maxScansToExplodeReached: false,
    winningPlan:
     { stage: 'PROJECTION_SIMPLE',
       transformBy: { actors: 1, title: 1, year: 1, _id: 0 },
       inputStage:
        { stage: 'COLLSCAN',
          filter: { actors: { '$eq': 'Chris Pratt' } },
          direction: 'forward' } },
    rejectedPlans: [] },
 executionStats:
  { executionSuccess: true,
    nReturned: 3,
    executionTimeMillis: 2,
    totalKeysExamined: 0,
    totalDocsExamined: 1000,
    executionStages:
     { stage: 'PROJECTION_SIMPLE',
       nReturned: 3,
       executionTimeMillisEstimate: 0,
       works: 1002,
       advanced: 3,
       needTime: 998,
       needYield: 0,
       saveState: 1,
       restoreState: 1,
       isEOF: 1,
       transformBy: { actors: 1, title: 1, year: 1, _id: 0 },
       inputStage:
        { stage: 'COLLSCAN',
          filter: { actors: { '$eq': 'Chris Pratt' } },
          nReturned: 3,
          executionTimeMillisEstimate: 0,
          works: 1002,
          advanced: 3,
          needTime: 998,
          needYield: 0,
          saveState: 1,
          restoreState: 1,
          isEOF: 1,
          direction: 'forward',
          docsExamined: 1000 } } },
  command:
   { find: 'movies',
     filter: { actors: 'Chris Pratt' },
     projection: { actors: 1, title: 1, year: 1, _id: 0 },
     '$db': 'SMD' }...
```

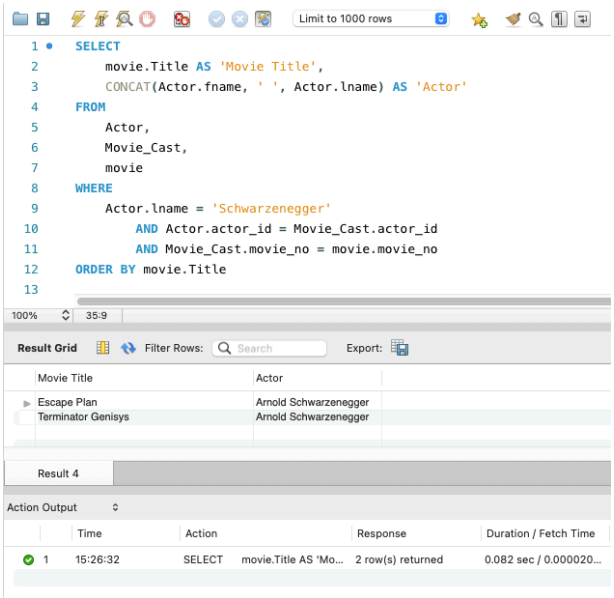

FIGURE 32. **SQL performance of query CU2 in MySQL Workbench.**

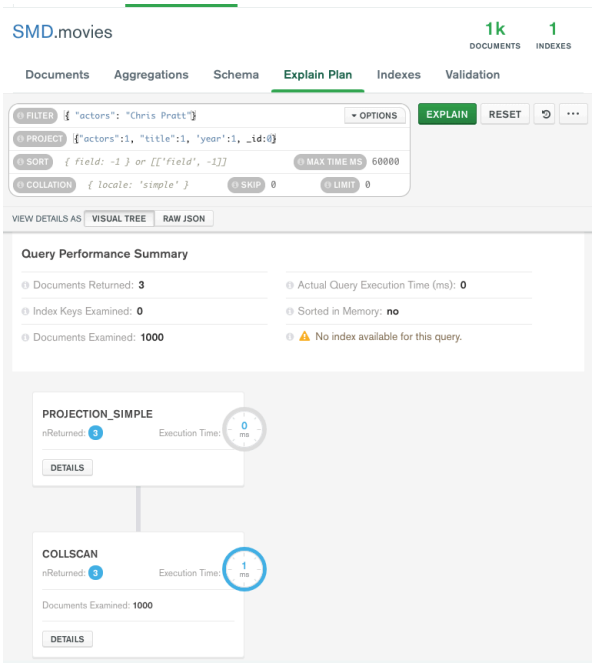

FIGURE 33. **NoSQL performance of query CU2 in MongoDB Explain Plan.**

Next, we illustrate performance measurement of an aggregation query, using query MA6. As seen in Figure 34, the duration time, or execution time, of this query in SQL was 0.087 seconds, or 87 milliseconds. In contrast, the execution of this query in NoSQL was approximately 3 milliseconds, as seen in the truncated execution stats below:

```
db.user.explain("executionStats").aggregate([{$project: {
 customer_service: 1,
 _id: 0}}, {$unwind: {
 path: '$customer_service'}}, {$match:
{'customer_service.created_at': {
$gte: '2021-07-01',
$lt: '2021-07-31'
 }}, {$group: {_id: '$customer_service.problem_desc',
NumberComplaintsJuly2021: {$count: {}}}}, {$sort:
{NumberComplaintsJuly2021: -1}}])
{ explainVersion: '1',
stages:
[ { '$cursor':
...
executionStats:
{ executionSuccess: true,
nReturned: 300,
executionTimeMillis: 3,
totalKeysExamined: 0,
totalDocsExamined: 300,
executionStages:
{ stage: 'PROJECTION_SIMPLE',
nReturned: 300,
executionTimeMillisEstimate: 2,
works: 302,
advanced: 300,
needTime: 1,
needYield: 0,
saveState: 1,
restoreState: 1,
isEOF: 1,
transformBy: { customer_service: true, _id: false },
inputStage:
{ stage: 'COLLSCAN',
nReturned: 300,
executionTimeMillisEstimate: 2,
works: 302,
advanced: 300,
needTime: 1,
needYield: 0,
saveState: 1,
restoreState: 1,
isEOF: 1,
direction: 'forward',
docsExamined: 300 } } } },
nReturned: 300,
executionTimeMillisEstimate: 2 },
{ '$unwind': { path: '$customer_service' },
nReturned: 300,
executionTimeMillisEstimate: 3 },
{ '$match':
{ '$and':
[ { 'customer_service.created_at': { '$gte': '2021-07-01' } },
{ 'customer_service.created_at': { '$lt': '2021-07-31' } } ] },
nReturned: 193,
executionTimeMillisEstimate: 3 },
{ '$group':
{ _id: '$customer_service.problem_desc',
NumberComplaintsJuly2021: { '$sum': { '$const': 1 } } },
maxAccumulatorMemoryUsageBytes: { NumberComplaintsJuly2021: 432 },
totalOutputDataSizeBytes: 1533,
usedDisk: false,
nReturned: 6,
executionTimeMillisEstimate: 3 },
{ '$sort': { sortKey: { NumberComplaintsJuly2021: -1 } },
totalDataSizeSortedBytesEstimate: 1629,
usedDisk: false,
nReturned: 6,
executionTimeMillisEstimate: 3 } ],
```

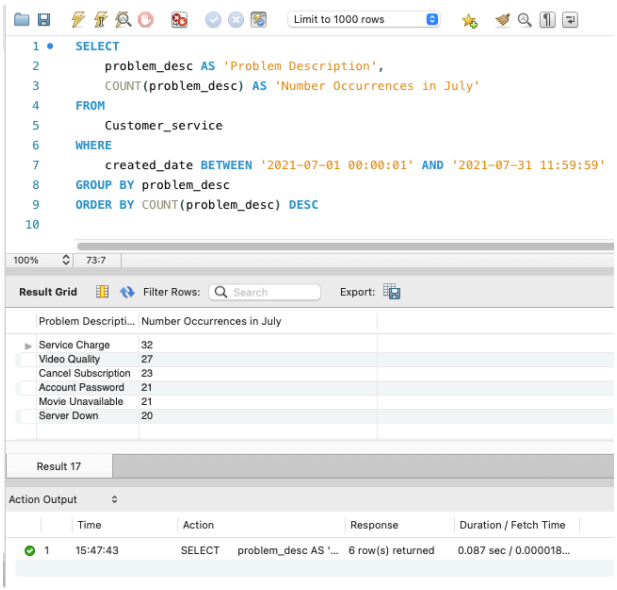

FIGURE 34. **SQL performance of query CU2 in MySQL Workbench.**

In Figure 35, we can see the performance comparison of common queries across the SQL and NoSQL databases.

Query Number	SQL Performance*	NoSQL Performance*
CUS1	84	10
CUS2	82	2
CUS3	83	2
CUS4	84	2
MA1	79	1
MA2	90	1
MA3	85	1
MA4	83	2
MA5	80	1
MA6	87	1
MA7	85	1
MA8	82	1

**The performance metric used is execution time in milliseconds.*

FIGURE 35. The table compares the performance of SQL and NoSQL databases, using milliseconds of execution time as the performance metric.

We can conclude from Figure 27 that NoSQL is much faster than SQL, primarily due to its denormalized nature. The query which took the longest execution time for NoSQL was CU1, where we joined both user and movies collections.

We also conclude from Explain Plan results that indexing, or sorting MongoDB collection in a particular order based on the value of one or more fields, would have sped up our queries, especially our find queries. However, the disadvantage of indexing is that it takes up memory, as seen in Figure 2. Since the collections of our database are relatively small in size, we have decided not to employ only one index per collection. Had our database been larger in size, we would have considered creating more indexes on various fields in the collections.

REFERENCES

[1] Analyze Query Performance - MongoDB. [Online]. Available: https://www.mongodb.com/docs/manual/tutorial/analyze-query-plan/. [Accessed: 15-April-2022].

[2] Explain Results - MongoDB. [Online]. Available: https://www.mongodb.com/docs/manual/reference/explain-results/#executionstats.

[3] Scheduled Triggers - MongoDB Realm. [Online]. Available: https://www.mongodb.com/docs/realm/triggers/scheduled-triggers/ . [Accessed: 15-April-2022].