UNIVERSITY OF VIENNA
INFORMATION MANAGEMENT AND SYSTEMS ENGINEERING

Milestone 2 Report

Samuel Šulovský
Michal Robert Žák

January 18, 2022

1 HOW TO RUN THE APP

To run the app, in the root directory run `docker-compose up --build` (or `docker-compose build` and then `docker-compose up`). After Docker finishes its set up (which should be noticeable by the fact that log messages come to a halt), navigate to `https://localhost:8443` and allow the self-signed certificate, so that the front end can communicate with the API. After that, you can connect to the web app at `https://localhost:443`, where you will also have to allow the self-signed certificate. The API will also, of course, be available at `https://localhost:8443` if you want to manually access it.

2 TECH STACK

On the database side, we used MySQL for our RDBMS and MongoDB for our NoSQL implementation. We used a Javascript based tech stack to communicate with the database and display data from it. Our application consists of a Node.js/Express[1] server that serves API endpoints, which the front end connects to to receive data using HTTP requests. The front end itself is a Next.js[2] web application. We used TypeScript to aid us in creating more robust, maintainable code both on the front end and the back end. We used the mysql2[3] and mongodb[4] packages to communicate with the databases from the back end. The front end sent requests and maintained state using mostly the React Query library[5], due to its automatic refetching, caching and overall simplification of network state handling.
We used data generated by Mockaroo for most data, except card game names and card types, the data for which we obtained from John McLeod at https://www.pagat.com/ with his explicit permission, attached in the appendix.

[1]https://expressjs.com/
[2]https://nextjs.org/
[3]https://www.npmjs.com/package/mysql2
[4]https://www.npmjs.com/package/mongodb
[5]https://react-query.tanstack.com/

2.1 Container Startup Time

Due to using Next.js for the front end, our container takes quite some time to build - sometimes multiple minutes (longest observed was 2.5 minutes). This is due to the limitations of docker running on low resources[6] and next compiler, which generates static pages as well as optimising them for production, so please be patient when running our container.

[6]https://github.com/vercel/next.js/discussions/17709

3 Data Migration

In this section, we will detail the logical and physical design of our RDBMS as well as how the migration to NoSQL was carried out and what the resulting tables looked like. Additionally, queries used to execute our use cases will be compared and demonstrated.

3.1 MySQL Database

The relational database is based on entities necessary to implement our desired functionality, such as Card Game or Card Type. The centerpiece of the diagram is the Review entity, our weak entity, which belongs to a Card Game and is left by a User. Card Games themselves use one of a select few Card Types and can become Verified Card Games upon being verified by Admins – Users who have been promoted by another Admin.
Compared to MS1 we decided to change the weak entity Review dependent on the Card Game entity instead of the User entity, as suggested in the feedback. Additionally, we changed the Age attribute in the User entity to the Birthday attribute, reflecting the change from age to a timestamp of the user's birth date instead.

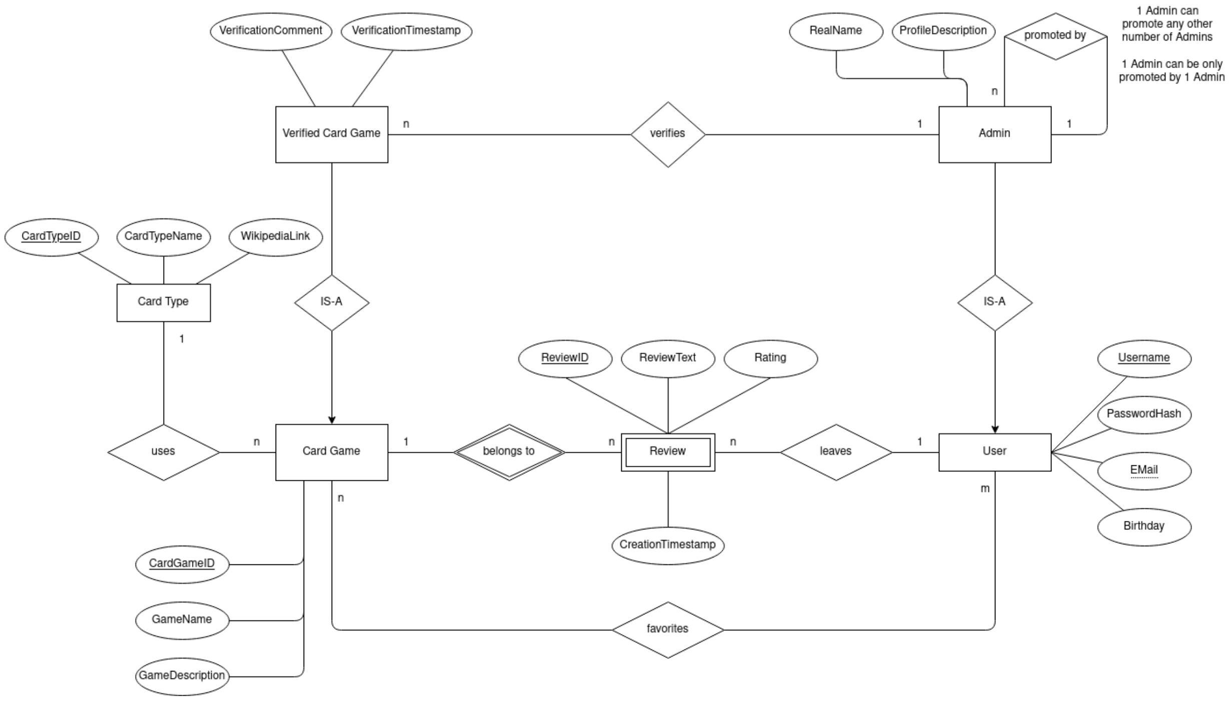


Figure 3.1: RDBMS ER Diagram

3.1.1 *Logical Design*

Below we have added the Logical Implementation, which was used as a basis for the Physical Implementation. All information contained in the logical design is also contained in the physical design, however the logical design is a lot easier to read.

Listing 1: RDBMS Logical Design

```
User(Username, PassswordHash, Email, Birthday)
          PK:       Username

Admin(Username, RealName, ProfileDescription, PromotedBy)
          PK:       Username
          FK:       Admin.Username <> User.Username
                    Admin.PromotedBy <> Admin.Username

CardType(ID, Name, WikipediaLink)
          PK: ID

CardGame(ID, Name, Description, CardTypeID)
          PK:       ID
          FK:       CardGame.CardTypeID <> CardType.ID

VerifiedCardGame(ID, Comment, Timestamp, VerifiedBy)
          PK:       ID
          FK:       VerifiedCardGame.ID <> CardGame.ID
                    VerifiedCardGame.VerifiedBy <> Admin.ID

Review(ID, CardGameID, LeftBy, ReviewText, Rating, Timestamp)
          PK:       ID, CardGameID
          FK:       Review.CardGameID <> CardGame.ID
                    Review.LeftBy <> User.Username

favorites(UserID, CardGameID)
          PK:       UserID, CardGameID
          FK:       favorites.UserID <> User.ID
                    favorites.CardGameID <> CardGame.ID
```

3.1.2 *Physical Design*

Some notable parts of the creation script (physical design) of our RDBMS are the autogenerating timestamps for inserted reviews, as well as unicode support for card game names. We made a number of assumptions about the type and shape of our data, which are mentioned in comments in the script. For example, we cap review lengths at 280 characters, which is the length of a tweet. We also make assumptions on the names of people, limiting them at 100 characters. Similarly, wikipedia links to card types are limited to 300 characters.

Listing 2: MySQL Database Creation Script

```sql
CREATE TABLE User (
  Username VARCHAR(18),
  PasswordHash BINARY(64), -- Assume SHA_512
  Email VARCHAR(254),
  Birthday DATE,

  PRIMARY KEY (Username)
);

CREATE TABLE Admin (
  Username VARCHAR(18),
  RealName VARCHAR(100), -- Assume most names under 100 char
  ProfileDescription VARCHAR(280), -- Same as a "Tweet"
  PromotedBy VARCHAR(18),

  PRIMARY KEY (Username),
  FOREIGN KEY (Username) REFERENCES User(Username) ON DELETE CASCADE,
  FOREIGN KEY (PromotedBy) REFERENCES Admin(Username) ON DELETE SET NULL
);

CREATE TABLE CardType(
  ID INTEGER AUTO_INCREMENT,
  Name VARCHAR(32), -- assumption
  WikipediaLink VARCHAR(300), -- assumption

  PRIMARY KEY (ID)
);

CREATE TABLE CardGame (
  ID INTEGER AUTO_INCREMENT,
  Name NVARCHAR(50), -- assumption; NVARCHAR for unicode
  Description TEXT, -- assumption
  CardTypeID INTEGER,

  PRIMARY KEY (ID),
  FOREIGN KEY (CardTypeID) REFERENCES CardType(ID) ON DELETE CASCADE
);


CREATE TABLE VerifiedCardGame (
  ID INTEGER,
  Comment VARCHAR(280),
  CreationTimestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  VerifiedBy VARCHAR(18), -- tweet length

  PRIMARY KEY (ID),
  FOREIGN KEY (ID) REFERENCES CardGame(ID) ON DELETE CASCADE,
  FOREIGN KEY (VerifiedBy) REFERENCES Admin(Username) ON DELETE SET NULL
);

CREATE TABLE Review (
  ID INTEGER AUTO_INCREMENT,
  CardGameID INTEGER,
  LeftBy VARCHAR(18),
```

```sql
    ReviewText VARCHAR(280), -- tweet length
    Rating TINYINT,
    CreationTimestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

    PRIMARY KEY (ID, CardGameID),
    FOREIGN KEY (CardGameID) REFERENCES CardGame(ID) ON DELETE CASCADE,
    FOREIGN KEY (LeftBy) REFERENCES User(Username) ON DELETE SET NULL
);

CREATE TABLE favorites (
    UserID VARCHAR(18),
    CardGameID INTEGER,

    PRIMARY KEY (UserID, CardGameID),
    FOREIGN KEY (UserID) REFERENCES User(Username) ON DELETE CASCADE,
    FOREIGN KEY (CardGameID) REFERENCES CardGame(ID) ON DELETE CASCADE
);
```

3.2 MongoDB Database

The MongoDB Database collection can be in our opinion best summarised by the diagram bellow. It should contain the most relevant information to give a good idea of the structure. All `id` parameters of the RDBMS design were replaced with `_id` parameters, using the ObjectID provided by MongoDB.

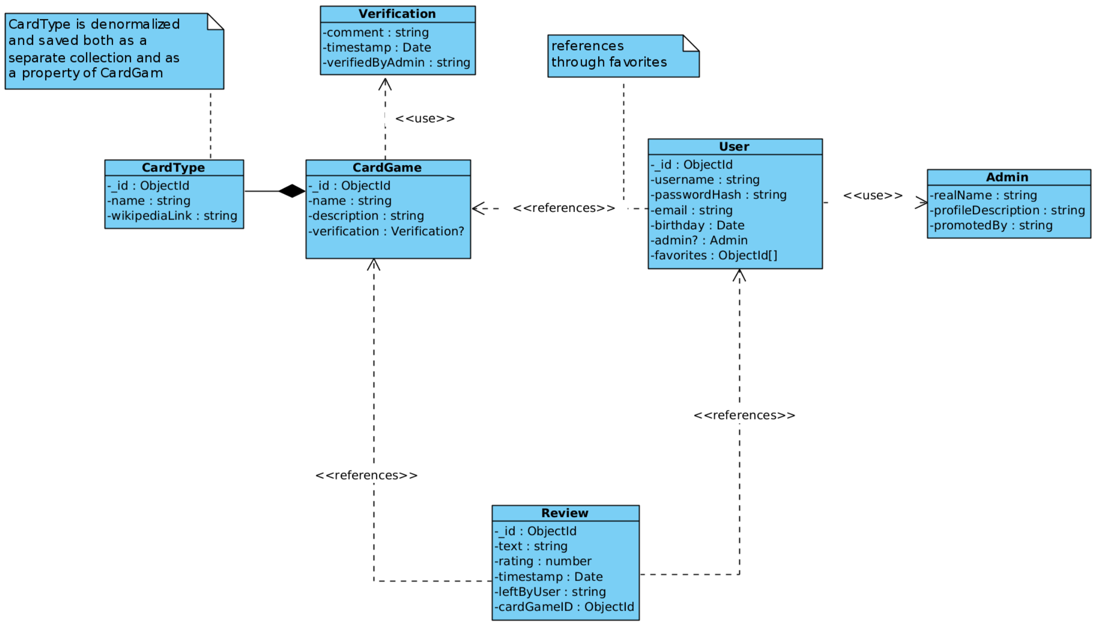


Figure 3.2: NoSQL Database Design

We used JavaScript to create our NoSQL tables using MongoDB, bellow you can see the document validation we created for each collection. This should give you an idea of how we structured our MongoDB documents.

3.2.1 CardGame

The CardGame entity was expanded with the Card Type, effectively joining the SQL tables Card Game and Card Type. Thus, Card Games contain identical BSONs to the Card Type collection within themselves. This optimises data fetching for card games, avoiding the join that was necessary in the SQL part. This was enabled by using the BSON object type, which let us define simple JSON-like objects to use as parameters.

Listing 3: Card Game Collection

```
db = new Mongo().getDB("card-game");

db.createCollection("cardGame", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      properties: {
        name: {
          bsonType: "string",
          description: "the name of the card game",
          required: true
        },
        cardType: {
          bsonType: "object",
          required: true,
          properties: {
            name: {
              bsonType: "string",
              description: "the name of the card type",
              required: true
            },
            wikipediaLink: {
              bsonType: "string",
              description: "the link to the wikipedia article of
                the specific card type",
              required: true
            }
          }
        },
        description: {
          bsonType: "string",
          description: "the description of the card game",
          required: true
        },
        verification: {
          bsonType: "object",
          required: ["comment", "timestamp", "verifiedByAdmin"],
          properties: {
            comment: {
              bsonType: "string",
              description: "verification comment from the admin",
            },
              timestamp: {
                bsonType: "timestamp",
                description: "the creation timestamp of the verification"
              },
              verifiedByAdmin: {
                bsonType: "string",
                description: "the username of the admin who verified
                  this card game"
              }
            }
          }
        }
      }
    }
});
```

The card game collection contains

3.3 CardType

The cardType collection was already visible inside the cardGame collection, however we have chosen to save it separately as well. This is due to the need to query the list of available cardTypes, which would be less than ideal if this data would need to be collected from cardGames. Furthermore the denormalization should not cause any problems as the CardType collection should seldom get updated or expanded. In fact adding or editing the CardType is not part of any use case, so we treat this data as immutable and essentially read-only. The fields contained in the documents saved in this collection are virtually identical to the SQL table.

Listing 4: Card Type Collection

```
db = new Mongo().getDB("card-game");

db.createCollection("cardType", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: ["name", "wikipediaLink"],
      properties: {
        name: {
          bsonType: "string",
          description: "the name of the card type"
        },
        wikipediaLink: {
          bsonType: "string",
          description: "the link to the wikipedia article of the specific
            card type"
        }
      }
    }
  }
});
```

3.4 Review

The Review collection is very similar to the corresponding SQL table, with the key difference being in the timestamp field, which can no longer be automatically generated. We have considered denormalizing this collection as well and saving (at least parts of it) in either the user or cardGame documents. This would have helped when we query for the report 2. In the end however we have ultimately decided against this as

1. each review document can grow very large as it contains a lot of text
2. a cardGame document could realistically grow to a comment size of over a 1000, even within a month.
3. a malicious user may spam our app with comments and thus increase the size of the reviews saved in his user document, throttling our performance once again.

To mitigate the cost of joining the data, we have added an index over the cardGameId table, to help better combine the cardGames and the reviews.

Listing 5: Review Collection

```
db = new Mongo().getDB("card-game");

db.createCollection("review", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      properties: {
        cardGameID: { bsonType: "objectId", required: true },
        leftBy: {bsonType: "string", required: true },
        reviewText: { bsonType: "string", required: true },
        rating: { bsonType: "number", required: true },
        creationTimestamp: { bsonType: "timestamp", required: true }
      }
    }
  }
});
```

3.5 User

The User collection is quite straightforward, though it's important to mention that the Admin table was completely removed and in its place the User collection got a new optional "admin" field which contains the Admin information for those users who are admins. Furthermore the favorites relation was realised as an array saved in the user document. Saving the favorites here should be alright as, even tough a user could technically favorite a lot of cardGames this is not the expected behaviour + the array consists of fairly small objects, namely the ids of the favorited cardGames, which should ensure this array does not explode in size. The User collection further has an index ensuring that the username of the user is unique.

Listing 6: User Collection

```
db = new Mongo().getDB("card-game");

db.createCollection("user", {
  validator: {
    $jsonSchema: {
```

```
    bsonType: "object",
    properties: {
      username: {
        bsonType: "string",
        description: "the username of the user",
        required: true
      },
      passwordHash: {
        bsonType: "string",
        description: "hashed and salted password",
        required: true
      },
      email: {
        bsonType: "string",
        description: "email of the user",
        required: true
      },
      birthday: {
        bsonType: "date",
        description: "the date the user was born",
        required: true
      },
      favorites: {
        bsonType: "array",
        required: true,
        items: {
          bsonType: "objectId",
          description: "the id of the favorited card game"
        }
      },
      admin: {
        bsonType: "objectId",
        required: ["realName", "profileDescription"],
        properties: {
          realName: {
            bsonType: "string",
            description: "the real name of the admin"
          },
          profileDescription: {
            bsonType: "string",
            description: "a short description of the profile"
          },
          promotedBy: {
            bsonType: "string",
            description: "the username of the admin who promoted
              this admin"
          }
        }
      }
    }
    }
  }
});

db.user.createIndex({"username":1}, {unique:true});
```

3.6 Comparison of queries

3.6.1 Add card game

Adding a card game is quite similar between MySQL and MongoDB, with the main difference being how data is inserted. For MySQL we need to write a SQL statement and add the values extracted from the object we received from the front end into the wildcards. While this isn't a prepared statement (the mysql2 JavaScript library can't use those) it does offer many benefits of prepared statements, such as preventing SQL injection attacks. MongoDB stores objects slightly differently to how the front end perceives them, so we use an extraction function to get the data we want from the received JSON and create a compliant object to be inserted.

MySQL

```
INSERT INTO CardGame(Name, Description, CardTypeId) VALUES (?, ?, ?)',
    [cardGame.name, cardGame.description, cardGame.cardType.id]
```

MongoDB

```
// We use a function to convert the JSON sent by the front end
// into an object able to be inserted into the Mongo DB
db.cardGame.insertOne(extractCardGame(cardGame));

export function extractCardGame(
    cardGame: CardGame,
    id?: ObjectId | undefined
  ) {
  if (!cardGame.verification) {
    return {
      _id: id,
      name: cardGame.name,
      cardType: {
        name: cardGame.cardType.name,
        wikipediaLink: cardGame.cardType.wikipediaLink
      },
      description: cardGame.description,
    }
  }

  const verification = !cardGame.verification ? undefined : {
    comment: cardGame.verification.comment,
    timestamp: cardGame.verification.timestamp,
    verifiedByAdmin: cardGame.verification.verifiedByAdmin
  };
  return {
    _id: id,
    name: cardGame.name,
    cardType: {
      name: cardGame.cardType.name,
      wikipediaLink: cardGame.cardType.wikipediaLink
    },
    description: cardGame.description,
    verification: verification
  }
}
```

3.6.2 Review card game

Leaving reviews is technically very similar to adding games, with the main difference being in how the timestamp is handled. While MySQL can automatically generate timestamps, these must be provided to MongoDB. We handle this by letting the back end create a timestamp when extracting the Mongo compliant object from the one the front end sent us.

MySQL

```
INSERT INTO Review(CardGameID, LeftBy, ReviewText, Rating)
VALUES (?, ?, ?, ?)', [cardGameId, review.leftByUser, review.text, review.rating];
```

MongoDB

```
// We use a function to convert the JSON sent by the front end
// into an object able to be inserted into the Mongo DB
db.review.insertOne(extractReview(review));

export function extractReview(review:Review) {
  return {
    cardGameID: review.cardGameId,
    leftBy: review.leftByUser,
    reviewText: review.text,
    rating: review.rating,
    // Auto generating a timestamp in JS, since Mongo can't
    creationTimestamp: !review.timestamp ? Date.now() : review.timestamp
  };
}
```

3.6.3 Get most reviewed card types in the last month

MySQL The SQL statement works in the following way:

1. Query all one month old reviews from the database
2. Join the reviews with the CardGame they belong to
3. Join the CardType of the CardGame with this joined table
4. Count the occurences for a given CardType name

The filter operation is done on the Review table as this should lead to a better performance, as less tables have to be joined and filtered later on.

```
SELECT CardType.Name AS CardTypeName, COUNT(RecentReview.ID) as ReviewCount FROM
   (SELECT * FROM Review
      WHERE Review.CreationTimestamp > (SELECT TIMESTAMP(
      DATE_SUB(NOW(), INTERVAL 30 day))) AS RecentReview
    LEFT JOIN CardGame ON RecentReview.CardGameID = CardGame.ID
    LEFT JOIN CardType ON CardGame.CardTypeID = CardType.ID
    GROUP BY CardType.Name ORDER BY COUNT(Review.ID) DESC;
```

MONGODB The MongoDB pipeline works in the following way:

1. select all `reviews` where the `creationTimestamp` is greater than the timestamp thirty days ago
2. join these `reviews` with the `cardGame` field over its `_id`
3. group the resulting join over the `cardType` names and count their occurrences
4. sort descending

```
db.review.aggregate([
  {$match: {creationTimestamp: {$gte: thirtyDaysAgo}}},
  {$lookup: {from: 'cardGame', localField: 'cardGameID',
    foreignField:'_id', as:'cardGame'}},
  {$group: {_id: '$cardGame.cardType.name', reviewCount: {$count: {}}}},
  {$sort: {reviewCount: -1}}
])
```

The MongoDB query statement should be better performing compared to the SQL one as the CardGame and CardType tables are prejoined and this operation does not need to be computed. Both queries perform a reduction before joining the data, which as far as we can tell should have roughly the same overhead. We have further decided to exclude those card types from our result, which have a total of 0 reviews. Both join operations of the statements should perform equally eficiently, as both join (lookup) over indexes.

3.6.4 Get card games popular with teens

MYSQL The SQL statement works in the following way:

1. join verified card games with card games and thus select only all verified card games
2. Join favorites to the verified card games
3. Select all users between the age of 13 and 18
4. Cross Join the VerifiedCardGames + favorites with the teen users
5. Count the occurrences of a particular CardGame name

The SQL statement filters the users before joining them to optimise the query a little.

```
SELECT CardGame.Name AS CardGameName,
COUNT(Teens.Username) as UserCount FROM VerifiedCardGame
LEFT JOIN CardGame ON VerifiedCardGame.ID=CardGame.ID
LEFT JOIN favorites ON CardGame.ID = favorites.CardGameID
CROSS JOIN (SELECT * FROM User WHERE User.Birthday >
(SELECT DATE(DATE_SUB(NOW(), INTERVAL 18 year)))
  AND User.Birthday < (SELECT TIMESTAMP(DATE_SUB(NOW(), INTERVAL 13 year))))
  AS Teens ON Teens.Username = favorites.UserID
GROUP BY CardGame.Name ORDER BY COUNT(Teens.Username) DESC;
```

MONGODB The MongoDB statement works in the following way:

1. select all users between the age of 13 and 18
2. for each in the array parameter favorites create a separate document with only this parameter as the favorites (kind of explode this array)
3. join the cardGame
4. filter out those documents that are not verified
5. count how often a given cardGame name occurs
6. sort descending

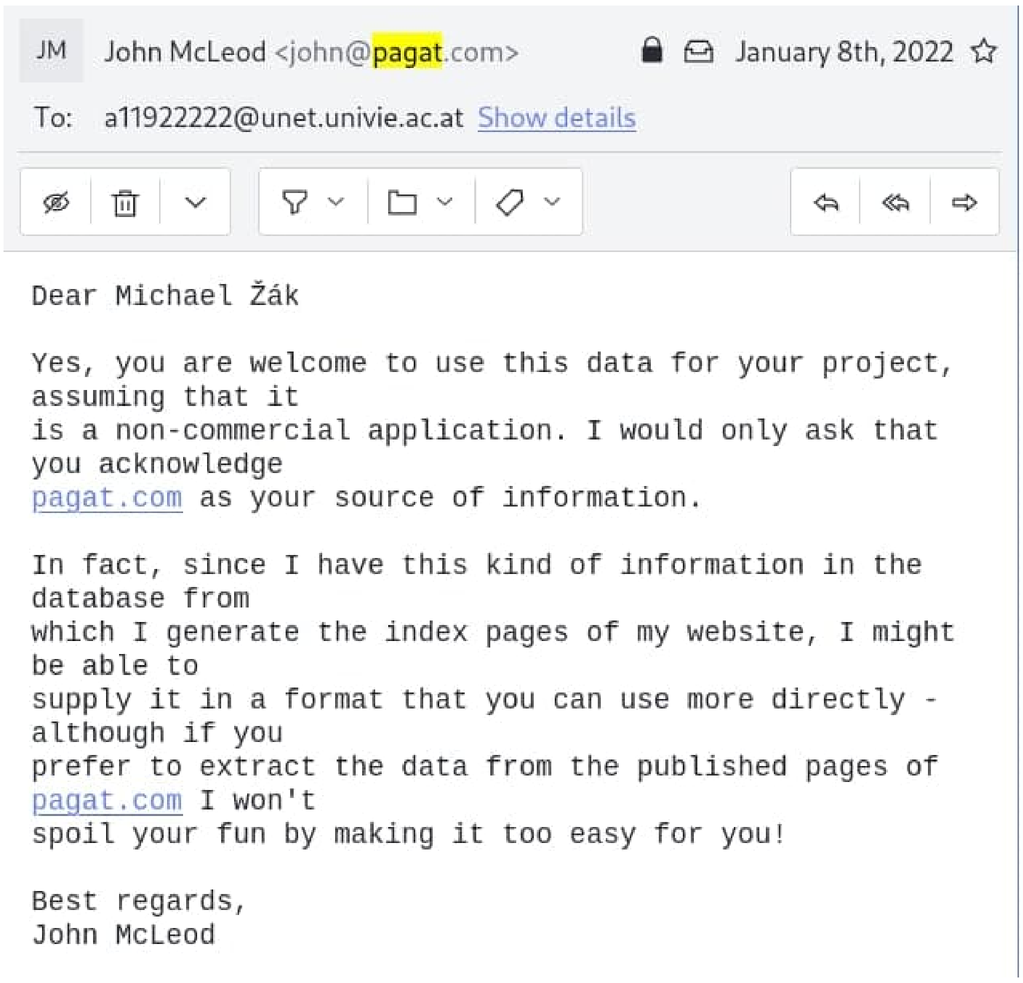
```
// We defined these helpers to use in the query
const today = Date.now();
const thirteenYearsAgo = new Date(today - 1000 * 60 * 60 * 24 * 365.25 * 13);
const eighteenYearsAgo = new Date(today - 1000 * 60 * 60 * 24 * 365.25 * 18)

// Mongo shell script
db.user.aggregate([
  {$match: {birthday: {
    $gte: eighteenYearsAgo,
    $lte: thirteenYearsAgo
  }}},
  {$unwind: {path: '$favorites'}},
  {$lookup: {from: 'cardGame', localField: 'favorites',
    foreignField:'_id', as:'cardGame'}},
  {$match: {"cardGame.verification": {$exists: true}}},
  {$group: {_id: '$cardGame.name', userCount: {$count: {}}}},
  {$sort: {userCount: -1}}
])
```

Comparing the statements we do not thing there will be major performance benefits to either. The only difference is that the verification is pre-joined in the MongoDb, whereas it has to be joined in SQL. This is however not that of a big win, as the MongoDb statement has to than find each entry where this verification is defined, whereas this happens automatically with the join operation in SQL.

4 Appendix

4.1 Permission to use card type and card name data from John McLeod



JM John McLeod <john@pagat.com> January 8th, 2022

To: a11922222@unet.univie.ac.at Show details

```
Dear Michael Žák

Yes, you are welcome to use this data for your project, assuming that it
is a non-commercial application. I would only ask that you acknowledge
pagat.com as your source of information.

In fact, since I have this kind of information in the database from
which I generate the index pages of my website, I might be able to
supply it in a format that you can use more directly - although if you
prefer to extract the data from the published pages of pagat.com I won't
spoil your fun by making it too easy for you!

Best regards,
John McLeod
```

Figure 4.1: Permission from John McLeod to use his data