UNIVERSITY OF VIENNA
INFORMATION MANAGEMENT AND SYSTEMS ENGINEERING

Milestone 2 Report

Daniel Petrov, Kyrylo Shevchenko

How to run the project

To run the project one must navigate to the root directory of the project. There should be a ‘docker-compose.yml’ file and after running the command ‘docker compose up’ a container is started. All the build work is done automatically by docker and only the necessary ports are being exposed.

The container consists of three running applications - frontend, backend which are Spring boot applications and a MySQL database. They are all isolated from one another and work in a shared network.

The backend can be accessed at ‘http://localhost:8080’. There one can find 2 buttons - one for filling the database with random data. If there’s already available data in the database it gets deleted and a new set is generated. The other button is for the best selling games report where one can filter based on genre and sort the games in ascending or descending order based on the total profit - game price * quantity of a purchased game across all orders.

The frontend can be accessed at ‘https://localhost:8443’ where one can interact with the web system and check out the use cases and the best rated game based on rating reviews report.

Tech Stack

The application itself contains 2 parts - backend and frontend.
For the backend part we used Java, together with the Java Spring Boot framework.

To communicate with the Frontend part we are using REST API`s and for the RDBMS part - MySQL database. We also used Java Hibernate to implement the JPA repositories and to map our objects into tables. To simplify the development process, we also used Lombok to generate getters, setters and other similar useful functionality.

Frontend was built using a simple HTML/CSS/JS stack with bootstrap for the design and jquery library to fetch the api endpoints and send requests to the backend.

For the secure access (HTTPS) on the frontend we used a self-signed certificate generated with the help of keytool - a utility tool provided by Java for managing keys and storing them in a keystore.

Everything regarding build-pipeline was implemented using docker and docker-compose files.

For effective cooperation, we used GitHub, to track the progress:
https://github.com/dzon337/gameshop

For faster and easier data generation we made use of the Javafaker library that can be found at "https://github.com/DiUS/java-faker".

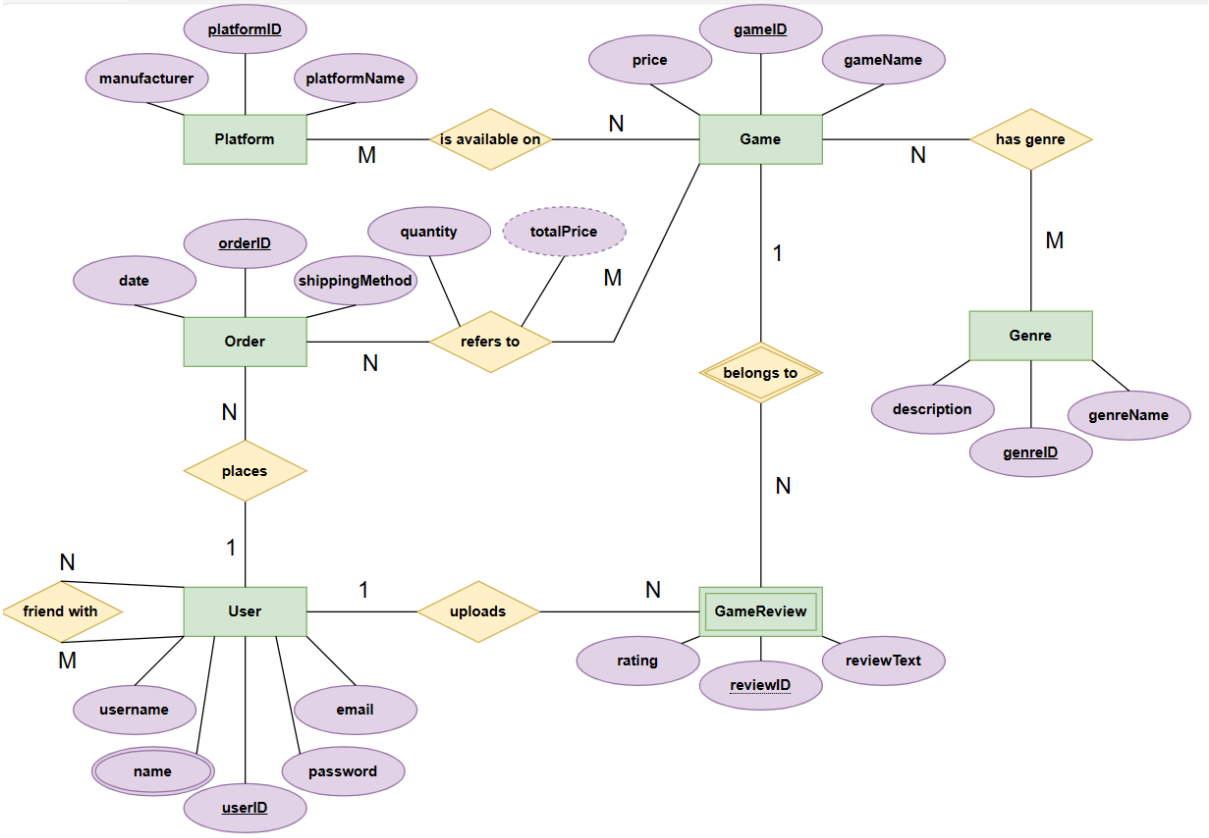
Changes made from Milestone 1

The GameDetails entity was removed from our original er-diagram as suggested in the feedback for milestone 1 and the leftover data was collected in the Game entity.

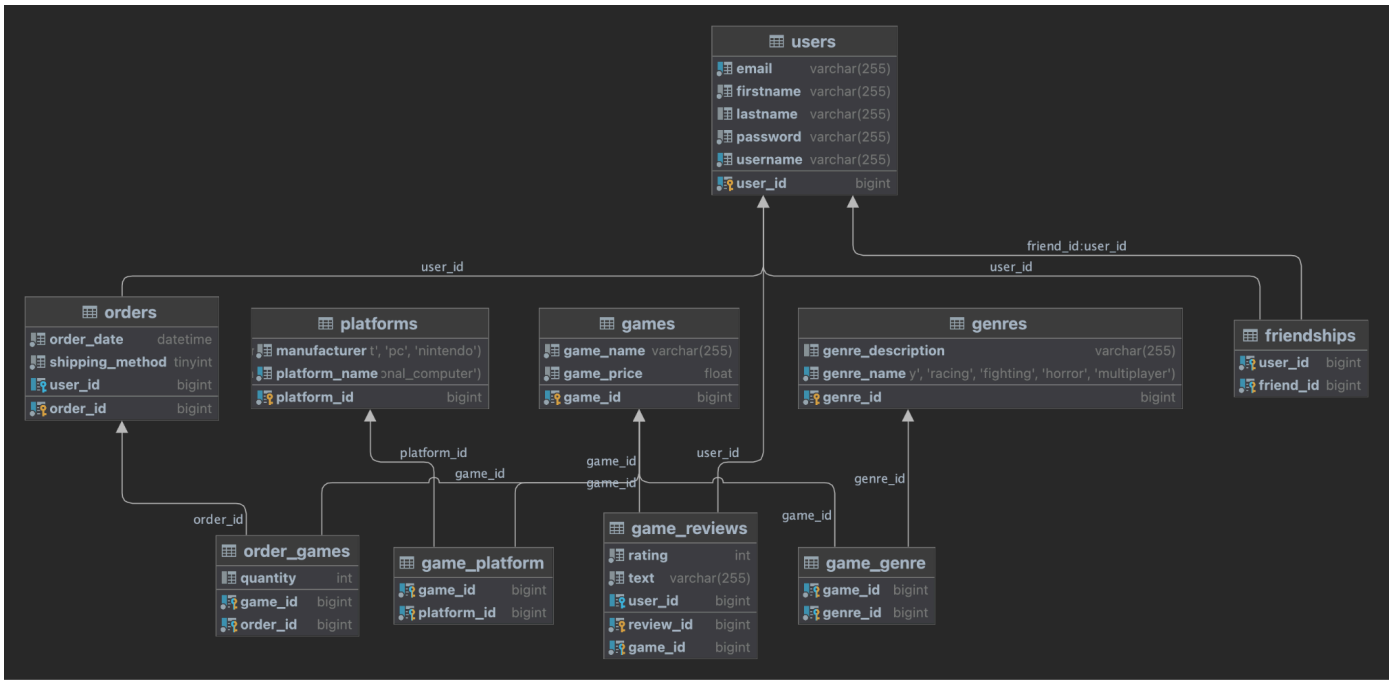
Also the following attributes were removed from Game - developer, release_year, description due to them not being relevant for our Gameshop project.

Furthermore, we replaced the rating attribute in Genre with a description.

Finally, the rating of games in a GameReview object now ranges from 1-10, unlike the one specified in our first main use case.



Physical MySQL design




```
CREATE TABLE games (
        game_id BIGINT AUTO_INCREMENT,
        game_name VARCHAR(255) NOT NULL UNIQUE,
        game_price FLOAT(23) NOT NULL,
        PRIMARY KEY (game_id)
);
```

```
CREATE TABLE genres (

        genre_id BIGINT AUTO_INCREMENT,

        genre_description VARCHAR(255),

        genre_name ENUM('SHOOTER', 'SPORTS', 'ADVENTURE',
'OPEN_WORLD', 'ACTION',     ' ROLE_PLAY', 'FIGHTING', 'MULTIPLAYER',
'HORROR', 'RACING') NOT NULL,

        PRIMARY KEY(genre_id)
);


 CREATE TABLE platforms (

        platform_id BIGINT AUTO_INCREMENT,

        manufacturer ENUM('SONY', 'MICROSOFT', 'PC', 'NINTENDO') NOT
NULL,

        platform_name ENUM('PLAYSTATION_5', 'PLAYSTATION_4',
'XBOX360', 'XBOX_SERIES_X', 'NINTENDO_SWITCH',
'Personal_Computer') NOT NULL,

        PRIMARY KEY(platform_ID)
);


CREATE TABLE game_genre (

        game_id BIGINT NOT NULL,

        genre_id BIGINT NOT NULL,

        PRIMARY KEY (game_id, genre_id),

        CONSTRAINT FKou7qtr2d8jvpemnniel386hgx FOREIGN KEY
(genre_id) REFERENCES genres (genre_id),
```

```
    CONSTRAINT FKj4fg0k0h6r6n8df60att283nj FOREIGN KEY
(game_id) REFERENCES games (game_id)

);


CREATE TABLE game_platform (

    game_id BIGINT NOT NULL,

    platform_id BIGINT NOT NULL,

    PRIMARY KEY (game_id, platform_id),

    CONSTRAINT FKrbhqd4lmddt2jmgdjy3kirt38 FOREIGN KEY
(platform_id) REFERENCES platforms (platform_id),

    CONSTRAINT FK4safgr5f71s67seckgpbttc1f FOREIGN KEY
(game_id) REFERENCES games (game_id)

);


CREATE TABLE game_reviews (

    review_id BIGINT NOT NULL,

    game_id BIGINT NOT NULL,

    user_id  BIGINT NOT NULL,

    rating INTEGER NOT NULL,

    review_text VARCHAR(255) NOT NULL,

    PRIMARY KEY (game_id, review_id),

    CONSTRAINT FK8wbl1jhvpws8ek1y4tfi0pv31 FOREIGN KEY
(game_id) REFERENCES games (game_id),
```

```
        CONSTRAINT FKndh5d3njmsvuo57vq3yu8sy9k FOREIGN
KEY(user_id) REFERENCES users (user_id)

);


CREATE TABLE orders (

        order_id BIGINT AUTO_INCREMENT,

        order_date DATE NOT NULL,

        user_id BIGINT NOT NULL,

        shipping_method ENUM('Express', 'Standard', 'International') NOT
NULL,

        PRIMARY KEY (order_id),

        CONSTRAINT FK32ql8ubntj5uh44ph9659tiih FOREIGN
KEY(user_id) REFERENCES users (user_id)

);

CREATE TABLE order_games (

        game_id BIGINT NOT NULL,

        order_id BIGINT NOT NULL,

        quantity INTEGER NOT NULL,

        PRIMARY KEY (game_id, order_id),

        CONSTRAINT FKm4b62t6qcx11dpm4xdg7j5l7q FOREIGN
KEY(game_id) REFERENCES games (game_id),

        CONSTRAINT FKr2igojb3rijgh6l8rm1bavoeo FOREIGN
KEY(order_id) REFERENCES orders (order_id)

);
```

```
CREATE TABLE users (
        user_id BIGINT AUTO_INCREMENT,
        email VARCHAR(255) NOT NULL,
        firstname VARCHAR(255) NOT NULL,
        lastname VARCHAR(255),
        password VARCHAR(255) NOT NULL,
        username VARCHAR(255) NOT NULL,
        PRIMARY KEY (user_id)
);

CREATE TABLE friendships (
        user_id BIGINT NOT NULL,
        friend_id BIGINT NOT NULL,
        PRIMARY KEY (user_id, friend_id),
        CONSTRAINT FKt0mh1j446gu5rqba17rnknuil FOREIGN KEY
(friend_id) REFERENCES users (user_id),
        CONSTRAINT FK4mcscxflf13uk72aupf6uwbgn FOREIGN KEY
(user_id) REFERENCES users (user_id)
);
```

Implementation of Web system

Daniel Petrov

Main use case: Upload a game review

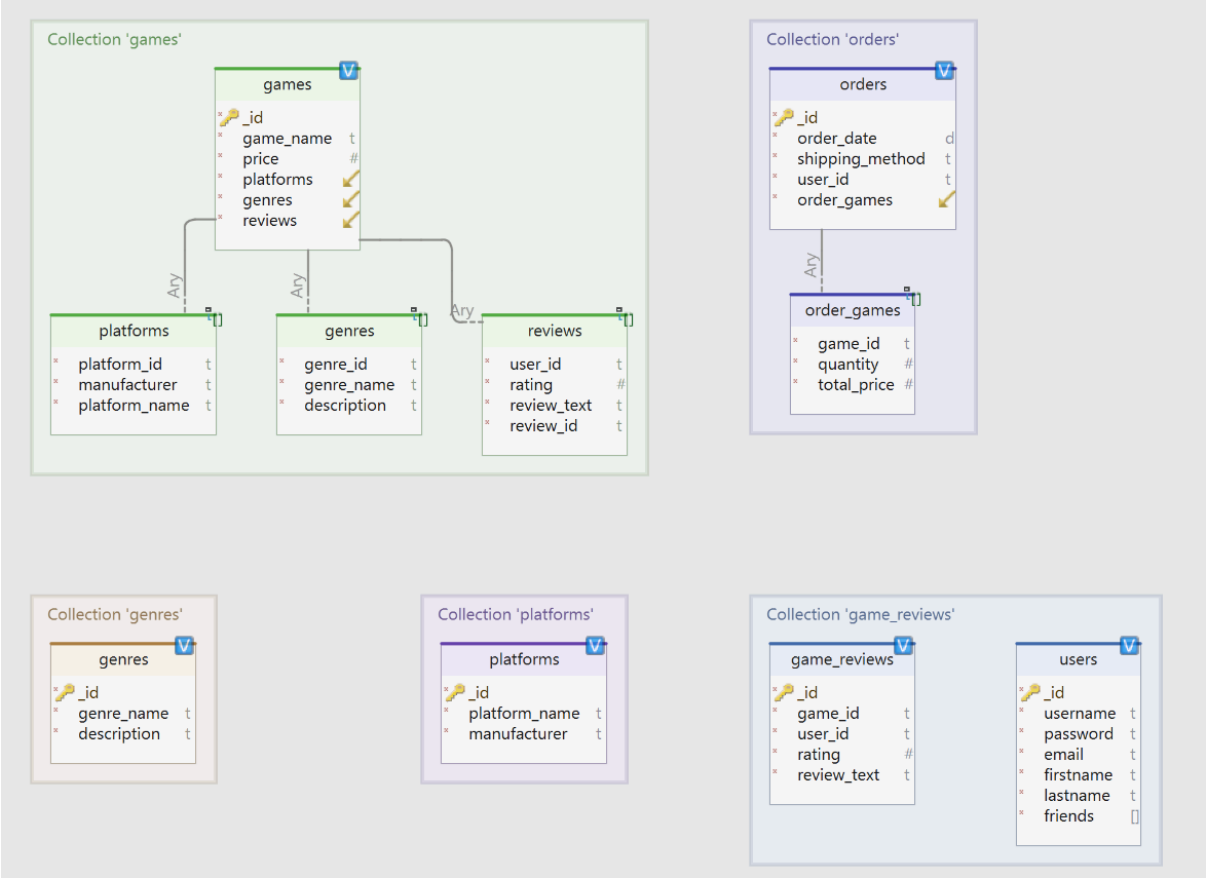
Report: Find out best selling games, filter them by genre and sort them by total profit

Kyrylo Shevchenko

Main use case: Register a new user

Report: Find out highest rated games based on review ratings and filter by genre and platform

NoSQL Design



genres_example_document = {

"_id": ObjectId("65a3c77c914a4188aaf4366e")

“genre_name”: “Action,

“description”: “Dive into high-octane action, wielding an arsenal of powerful firearms to outmaneuver and out shoot opponents in fast-paced environments.”

}

platforms_example_document = {

"_id": ObjectId("65a3c79b914a4188aaf43670"),

“platform_name”: “Playstation 5”,

“manufacturer”: “Sony”

}

The game_reviews collection is almost identical to the game_reviews table in MySql with the difference being that in this case we actually don’t need a “review_id” attribute to achieve uniqueness because MongoDb by default generates an “_id” attribute which serves the role of a unique primary key. We still need the “game_id” though to be able to find out to which exact game this review refers to.

game_reviews_example_document = {

"_id": ObjectId(“65a3cc4e914a4188aaf43688")

"game_id": “65a3c8f6914a4188aaf43679”,

"user_id": “65a3c813914a4188aaf43675”,

"rating": 10,

"review_text": "Amazing game!"

}

For the orders collection we decided to embed some data. In the case of the order_games table in MySql there was a game_id, order_id which is identical to a specific order entry and quantity. In the case of NoSql this table basically becomes redundant so embedding is, according to us, the better alternative. Additionally, everytime one accesses an order one is also interested in the games ordered and their quantity to determine the total price of a given order which is also relevant for the best selling games report. We store the reference object of the game “_id” to avoid redundancy and the usage of extra disk space and additionally save an attribute called “total_price” whose value is based on the quantity and price of a particular game. We don’t expect an order to consist of a big number of games.

order_example_document = {

“_id”: ObjectId(“65a3d76b914a4188aaf436c6”),

“order_date”: {

“$date”: “2023-10-31T00:00:00Z”

},

“user_id”: “65a3c813914a4188aaf43675”

“shipping_method”: “Express”,

“order_games”: [

{

“game_id”: “65a3c8f6914a4188aaf43679”,

“quantity”: 2,

“total_price”: 79.98

}

]

}

For the games collection we decided to fully embed the platforms, genres and reviews. For the genres and reviews we hardly expect any changes in their size and or value to be updated so embedding is not bad in this case. For the reviews we expect a higher frequency of adding new reviews but still expect them to be in the range of O(100) - O(1000). Also when one lands on the main page the games and all of their relevant information should be displayed together so by storing the data embedded like this we basically make sure that we only need one read operation for each game to be fetched and we avoid join operations of course by sacrificing some disk space and creating redundancy.

games_example_document = {

“_id”: ObjectId("65a3c8f6914a4188aaf43679"),

“game_name”: “GTA: V”,

“price”: 39.99,

"platforms": [

{

"platform_id": “65a3c79b914a4188aaf43670”,

"manufacturer": "Sony",

"platform_name": "Playstation 4"

```
        }
    ],
    "genres": [
        {
            "genre_id": “65a3c77c914a4188aaf4366e”,
            "genre_name": "Action",
            “description”: “Dive into high-octane action, wielding an arsenal of powerful firearms to outmaneuver and out shoot opponents in fast-paced environments.”
        }
    ],
    "reviews":  [
            {
                “review_id”: “65a3cc4e914a4188aaf43688”,
                “user_id”: “65a3c813914a4188aaf43675”,
                "rating": 10,
                "review_text": "Amazing game!”
            }
    ]
}
```

Finally for the users collection - the structure is almost identical but here there’s an array that stores the IDs of friends. We never actually need to access the actual information about a specific friend when we work with the user. In one of our use cases we should add another existing user. So to do that we need to check if this other person exists and if we are not already friends. We can achieve this by just using the ID, not the entire object. For that reason it makes much more sense to just store the id as a reference. Also we don’t expect a user to have more than 100 or 200 friends.

user_example_document = {

```
    “_id”: ObjectId(“65a3c813914a4188aaf43675”),
    “username”: “Flapjack”,
    “email”: “Flapjack@email.com”,
    “firstname”: “Flapjack”,
```

“lastname”: “Jackflap”,

“password”: “Flapjacks’ very safe password”,

“friends”: [“65a3c7c0914a4188aaf43674”]

}

SQL to NoSQL comparison

Register User

NoSQL:

```
db.users.insertOne({
        firstName: "John",
        lastName: "Doe",
        email: "johndoe@example.com",
        username: "johndoe",
        password: "password123",
        friends: []
});
```

SQL:

```
INSERT INTO users (first_name, last_name, email, username, password)
VALUES ('John', 'Doe', 'johndoe@example.com', 'johndoe', ‘password123');
```

Write a review

NoSql:

```
db.game_reviews.insertOne({
        _id: “65a3cc4e914a4188aaf43688”,
```

```
        game_id: “65a3c8f6914a4188aaf43679”
        rating: 10,
        review_textt: ”Best game ever”,
        user_id: “65a3c813914a4188aaf43675“
});

db.games.updateOne({
        { _id: “65a3c8f6914a4188aaf43679” },
        $push: {
                reviews: {
                        “review_id”: “65a3cc4e914a4188aaf43688”,
                        “user_id”: “65a3c813914a4188aaf43675”,
                        "rating": 10,
                        "review_text": "Amazing game!”
                }
        }
});
```

SQL:

```
INSERT INTO game_reviews (review_id, rating, review_text, game_id, user_id)
VALUES (999, 7, “Good review example”, 3, 420);
```

Add another user as a friend

NoSQL:

```
db.users.update(
        { username: "Flapjack" },
                {
                        $push: {
                                friends: ObjectId("0z21l2mz514622ed49d10d2f")
                         }
```

```
                }
);
```

SQL:

```
INSERT INTO friendships (user_id, friend_id) VALUES (29, 119);
```

Purchase a game

NoSQL:

```
db.orders.insertOne({
        order_date: ISODate("2023-10-31T00:00:00z"),
        shipping_method: "Express",
        user_id: "65a3c813914a4188aaf43675"
        order_games: [
                { "game_id": "65a3c8f6914a4188aaf43679", quantity: 2, total_price: 79.98 }
        ]
});
```

SQL:

```
INSERT INTO orders (order_date, shipping_method, user_id) VALUES (NOW(), 1, 12345);
INSERT INTO order_games (quantity, game_id, order_id)
VALUES  (2, 101, LAST_INSERT_ID()),  (1, 102, LAST_INSERT_ID()),
                                                (3, 103, LAST_INSERT_ID());
```

Bestseller games Report

MySql Query:

“SELECT g.game_name, g.game_price, ge.genre_name, SUM(og.quantity) AS sold_copies, SUM(og.quantity * g.game_price) AS total_profit FROM orders o JOIN order_games og ON o.order_id = og.order_id JOIN games g ON g.game_id = og.game_id JOIN game_genre gg ON gg.game_id = g.game_id JOIN genres ge ON ge.genre_id = gg.genre_id GROUP BY g.game_name, g.game_price, ge.genre_id ORDER BY total_profit DESC;”

MongoDB Query:

```
db.orders.aggregate([

	{ $unwind: "$order_games" },

		{
			$lookup: {
				from: "games",
				localField: "order_games.game_id",
				foreignField: "_id",
				as: "game"
			}
	},

	{ $unwind: "$game" },

		{
			$lookup: {
				from: "genres",
				localField: "game.genres.genre_id",
				foreignField: "_id",
				as: "genre"
			}
	 },
```

```
        { $unwind: "$genre" },

                {
                        $group: {
                                _id: "$game._id",
                                game_name: { $first: "$game.game_name" },
                                game_price: { $first: "$game.price" },
                                genre_name: { $first: "$genre.genre_name" },
                                sold_copies: { $sum: "$order_games.quantity" },
                                total_profit: { $sum: { $multiply: ["$order_games.quantity",
"$game.price"] } }
                        }
        },

        { $sort: { total_profit: -1 } }
]);
```

By denormalizing the data in MongoDB we make use of data locality, reduce the amount of joins operations by 3 - in the MySql query we perform 5 join operations which can be very costly and can hinder performance and make use of indexes.

Highest Rated games Report

MySql Query:

```
SELECT G.GameID, G.GameName, G.GamePrice, Ge.GenreID,Ge.GenreName,
P.PlatformID, P.PlatformName, AVG(GR.Rating) as AverageRating
FROM
   Game G
JOIN
   Genre Ge ON G.GenreID = Ge.GenreID
JOIN
   Platform P ON G.PlatformID = P.PlatformID
JOIN
   GameReview GR ON G.GameID = GR.GameID
WHERE
   Ge.GenreName = 'Specific Genre' AND
   G.GamePrice <= SpecificPrice AND
```

```
    P.PlatformName = 'Specific Platform'
GROUP BY
    G.GameID, G.GameName, G.GamePrice, Ge.GenreID, Ge.GenreName,
P.PlatformID, P.PlatformName
ORDER BY
    AverageRating DESC;
```

MongoDB Query:

```
db.Game.aggregate([
    {
        $lookup: {
            from: "GameReview",
            localField: "GameID",
            foreignField: "GameID",
            as: "reviews"
        }
    },
    {
        $lookup: {
            from: "Genre",
            localField: "GenreID",
            foreignField: "GenreID",
            as: "genre"
        }
    },
    {
        $lookup: {
            from: "Platform",
            localField: "PlatformID",
            foreignField: "PlatformID",
            as: "platform"
        }
    },
    {
        $match: {
            "genre.GenreName": "Specific Genre",
            "GamePrice": { $lte: SpecificPrice },
            "platform.PlatformName": "Specific Platform"
        }
    },
    {
        $unwind: "$reviews"
    },
```

```
    {
        $group: {
            _id: "$GameID",
            GameName: { $first: "$GameName" },
            GamePrice: { $first: "$GamePrice" },
            GenreID: { $first: "$genre.GenreID" },
            GenreName: { $first: "$genre.GenreName" },
            PlatformID: { $first: "$platform.PlatformID" },
            PlatformName: { $first: "$platform.PlatformName" },
            AverageRating: { $avg: "$reviews.Rating" }
        }
    },
    {
        $sort: {AverageRating: -1}
    }
])
```

NoSql Indexes

This index ensures fast lookups when searching for games with a specific genre.

```
db.genres.createIndex({genre_name: 1}, {unique: true})
```

Allows for easier and faster querying of games based on platform when filtering.

```
db.platforms.createIndex({platform_name: 1}, {unique: true})
```

Assures that each user can upload only one review about a game preventing duplicates. Before inserting one must find if a given user has already uploaded a review about this particular game.

```
db.game_reviews.createIndex({ game_id: 1, user_id: 1 }, { unique: true });
```

Allows for faster filtering of games based on chosen platforms and genres.

```
db.games.createIndex({ "platforms.platform_id": 1 });
db.games.createIndex({ "genres.genre_id": 1 });
```

These two indexes make it faster to lookup a user based on chosen username to avoid duplicate users and checking to see if a given person is already in the friends list.

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

Conclusion

MongoDB is optimized for queries on document-based, denormalized data. Complex joins are less common as related data is often embedded within a single document. Regarding MongoDB, it uses less tables in our case so the data is not so spread across the database. It reduces the number of joins, which reduces the complexity of the queries and provides a better overall performance for simple requests.

A lot of table data is already embedded in MongoDB, for example User entity. Relationships (like friends, reviews, orders) are typically stored in separate tables and linked using foreign keys. This normalizes the data but requires JOIN operations to aggregate data from multiple tables. Same goes for the Order table, MongoDB document simplifies the readability of the data and improves the efficiency of the database.

Regarding “Purchasing a game” , in MongoDB you can see a reduced number of queries to perform the operation : 1 instead of 2. We used 2 SQL insertions for it.
When you fetch a document, all the embedded data is retrieved in a single I/O operation. This is particularly beneficial for read-heavy applications where you often need to access related data together.

MongoDB is designed for scalability, particularly horizontal scaling. Denormalization is well-suited to this model because it often reduces the need for cross-node JOIN operations.

Explanation of Design Choices

1. Embedding vs. Referencing:
- Reviews are embedded within games to reduce read operations for fetching game details along with reviews. This is especially beneficial for the use case of viewing game details along with user reviews.
- Orders include references to games rather than embedding game details directly, as orders can be large and frequently updated. Embedding game details could lead to large documents and potential performance issues.

2. Indexing:
- Indexes are chosen based on query patterns. For instance, indexing `username`, `email`, and `game_id` allows for efficient lookups and is beneficial for the use cases of registering a user and uploading a review.
- For the report queries, indexing fields like `order_id`, `userId`, `gameOrders.gameId`, and `game_name` ensures efficient aggregation and sorting.

3. UserId in Game Reviews:
- Including `userId` in each review within the games collection allows for directly associating reviews with users. This is useful for the use case of a user uploading a review, as it provides a direct reference to the reviewer.

4. Composite Key in GameReview:
- In MongoDB, each document must have a unique `_id`. While we use `_id` as the primary key (default in MongoDB), it's also possible to create a composite key using `review_id` and `game_id` for relational consistency. However, MongoDB's document model typically relies on the unique `_id` field for simplicity and efficiency.

It leverages strengths in handling large and nested data structures and reduces the need for complex joins.