

**BC2402: DESIGNING & DEVELOPING DATABASES**

**AY19/20 SEMESTER 1**

**GROUP PROJECT REPORT**

**CHOONG HAN YI - U1820140J**

**JARED ANG RUI XIANG - U1811611A**

**LEE MUN YEE - U1810810A**

**NG JING XUN - U1820959A**

**TAN JING YU, DENIS - U1810101D**

**TAN WEI LUN - U1822950L**

[**1. Executive Summary**](#_jyz7e8q0b00u) **4**

[**2. Relational Data Model**](#_uok8bdggd1dq) **5**

[2.1 Entity Relationship Diagram (ERD)](#_sg6c34m76hyi) 5

[2.2 Improvements made on MHL’s ERD](#_aubb3oudod2) 6

[2.3 Key Assumptions](#_net8e7to24h3) 8

[**3. Relational Database Normalization**](#_dlqyz71upbay) **9**

[3.1 First Normal Form (1NF)](#_34ertjdw8gt1) 9

[3.2 Second Normal Form (2NF)](#_4m3sixt62uvy) 9

[3.3 Third Normal Form (3NF)](#_kp7glom8sm2m) 9

[3.4 Boyce-Codd Normal Form (BCNF)](#_5yykslna6x3z) 10

[**4. Relational Schema**](#_wvy0f3xma9pd) **11**

[**5. Non-Relational Data Model**](#_anoiuiruzbff) **12**

[5.1 Choice of NoSQL Database Program](#_khzcxk7p0l3z) 12

[5.2 JSON Attribute - Value Structure](#_64tmqm1foryc) 13

[5.2.1 User Collection](#_d8wjj1n9ylqb) 14

[5.2.2 Challenge Collection](#_g4azxjozjicd) 15

[5.2.3 Club Collection](#_reg8z169fl4s) 15

[**6. Recommendations to MHL**](#_tkcra1zgcg3z) **16**

[**7. Appendix**](#_uh5gtacybcl2) **19**

[7.1 Relational Database Deployment in MySQL](#_dviczpexum6m) 19

[7.2 Non-Relational Database Deployment in MongoDB](#_1e93f18xbze7) 19

# **1. Executive Summary**

This project focuses on exploring the options of backend data storage in the MHL Fitness 365 mobile application. During the design and development phase, our team has implemented both a relational and non-relational data model for a set of randomly-generated test data which closely models real-life use cases of the mobile application.

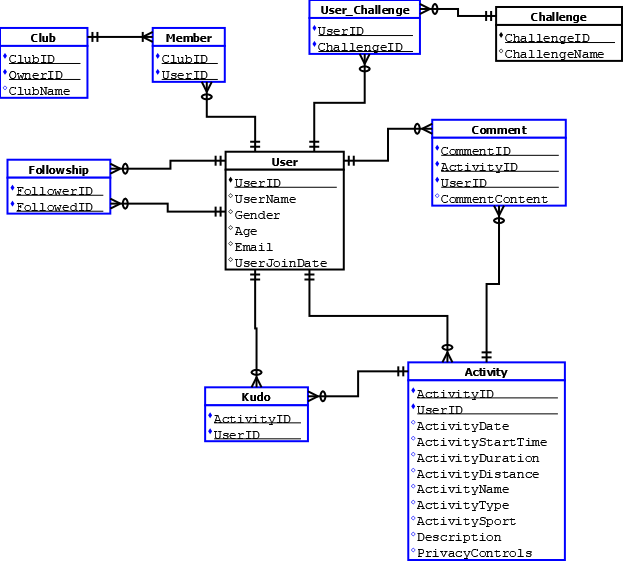
A relational database management system (RDBMS) was chosen due to its strict compliance with the ACID (Atomicity, Consistency, Isolation, Durability) properties, which are all characteristics that are highly desired in any database model.

There are, however, drawbacks to a rigid schema frequently encountered in a RDBMS, specifically the lack of horizontal scalability, data partitioning capabilities, and inflexibility to changing schemas. These are all flaws which are well-addressed by a non-relational database management system, which we have subsequently designed and implemented.

Finally, with two data models to compare and contrast, we shall propose recommendations and suggestions to the preferred data model that MHL is advised to use for the purpose of the Fitness 365 mobile application.

# **2. Relational Data Model**

## **2.1 Entity Relationship Diagram (ERD)**



**Fig. 1 Entity Relationship Diagram**

## **2.2 Improvements made on MHL’s ERD**

From the preliminary relational schema and entity-relationship diagram (ERD) provided by MHL, our team has made the following improvements and modifications, displayed in the revised ERD in Fig. 1.

Throughout the improvement process, key database design considerations of minimizing data redundancy, enforcing referential integrity, and performing database normalization have been carefully followed, to ensure ACID-compliance of our relational data model.

The following lists the improvements we have made from the original data model provided by MHL, and the rationale for taking these steps.

**Improvement 1 : Decomposition of many-to-many (M:N) relationships**

* User (Recursive Relationship) via the associative entity, *Followship*
* User and Activity, through the use of a composite primary key, containing both ActivityID and UserID. This is because we believe that the initial relationship was wrongly represented because an activity cannot be shared among many users, instead each user can have many activities
* User and Challenge, via the associative entity, *User-Challenge*
* User and Club, via the associative entity, *Member*

By decomposing the M:N relationships present in the initial model, data duplication and redundancy is avoided, thereby improving query performance. Referential integrity is also ensured via the decomposition process. The associative entity, *Member*, further enforces the assumption that a club’s owner (identified by the OwnerID), must also be a member of the club (identified by the corresponding UserID).

**Improvement 2 : Removal of derived attributes from entities**

* ***Activity* :** ActivityPace can be derived by dividing ActivityDistance from ActivityDuration
* ***PersonalBest* :** All attributes within the PersonalBest entity can be calculated or derived from attributes of the other entities.
* ***WeeklyLeaderboard* :** All attributes within the WeeklyLeaderboard entity can be calculated or derived from attributes of the other entities.

Storing derived attributes can result in costly database writes when these attributes are modified and updated, to ensure data consistency and synchronisation, leading to additional delays and performance lags. Our team hence advises derived data to be calculated on the fly, whenever needed, to significantly improve database performance, especially when working with large datasets.

**Improvement 3: Correction of inappropriate relationships between entities**

* **Activity and User :** We changed the original many-to-many relationship to a one-to-many relationship, as every activity is uniquely created by one user, and hence cannot be shared among multiple users. On the other hand, each user can create multiple activities. Hence, a one-to-many relationship will be more appropriate.
* **Activity and Comment :** We altered the original one-to-one relationship to a one-to-many relationship, as each activity can receive multiple comments, and each comment belongs to exactly one activity.
* **User and Comment :** We altered the original one-to-one relationship to a one-to-many relationship, as each user can make multiple comments on an activity, and each comment each comment belongs to exactly one user.
* **Activity and Kudo :** We altered the original one-to-one relationship to a one-to-many relationship, as each activity can receive multiple Kudos, and each Kudo belongs to exactly one activity.
* **User and Kudo :** Similarly, we altered the original one-to-one relationship to a one-to-many relationship, as each user can give multiple Kudos, and each Kudo belongs to exactly one user.

These relationships provide a more accurate real-life representation of the relationships between the entities, and hence allow us to ensure the reliability of our relational data model in modelling real-life data flow.

## **2.3 Key Assumptions**

During the development and improvement of our ERD to the final version shown in Fig. 1, our team has made the following important assumptions regarding the generation of data within the mobile application, as well as key constraints that we have imposed.

* A club owner must also be a member of the club. In other words, upon club creation by a user X, X will automatically be added to the associative entity *Member*, to reflect his membership in the club that he has just created.
* A user can make multiple comments and can have multiple comments on an activity.

# **3. Relational Database Normalization**

After the completion of our entity-relationship diagram, we shall now create our relational schema. Before we can do that, however, we have to ensure every relation in our schema obeys database normalization principles, minimally until the Third Normal Form (3NF). We have included a further normalization to the Boyce-Codd Normal Form (BCNF) for stricter compliance of our developed RDBMS to industry design principles.

## **3.1 First Normal Form (1NF)**

Since each attribute in every entity is atomic and discrete, no further modifications need to be done. 1NF has already been satisfied.

## **3.2 Second Normal Form (2NF)**

There exists partial dependency in the Comment entity, where CommentContent depends only upon CommentID, which is only part of the composite primary key of the Comment entity. To solve this problem, we must create a new entity called Comment-Content. The primary key of this new entity, CommentID, will be inherited from the Comment entity. The CommentID also enforce referential integrity between both entities. The attribute, CommentContent, will be removed from the Comment entity and instead be placed in the new entity. Now, the CommentContent is fully functionally dependent on the primary key CommentID. Thus, 2NF has been satisfied.

## **3.3 Third Normal Form (3NF)**

Transitive dependency, referring to a non-prime attribute deriving another non-prime attribute, is non-existent. As this is only applicable to relations with at least 2 non-key attributes, we shall only check the User and Activity entities,for which the requirement is already satisfied.

## **3.4 Boyce-Codd Normal Form (BCNF)**

Although not an industrial norm expected of most relational database management systems today, our team has chosen to perform normalization up till this standard, to ensure the strictest conformation of our RDBMS to modern database design principles, minimizing the possibility of errors and bugs developing during the course of use. To achieve a BCNF level of normalization, we checked each and every of our entities for the following anomaly which violates BCNF : *A non-prime attribute deriving a prime attribute.* After making a few reasonable assumptions and analyzing multiple real-life use case scenarios, we have concluded that our relational schema conforms and meets the requirements as specified in BCNF, thereby ensuring that it is BCNF-compliant.

# **4. Relational Schema**

With the ERD completed and the database normalized to BCNF, we can now develop a relational schema for our relational data model. The implementation was performed in MySQL, where primary, foreign key, and data type constraints were implemented strictly enforced to ensure referential integrity.

* User-Challenge[**(UserID)**, **(ChallengeID)**]Activity[**ActivityID**, **(UserID)**, ActivityDate, ActivityStartTime, ActivityDuration, ActivityDistance, ActivityName, ActivityType, ActivitySport, Description, PrivacyControls]
* Challenge[**ChallengeID**, ChallengeName]
* Club[**ClubID**, **(OwnerID)**, ClubName]
* Comment[**CommentID**, **(ActivityID)**, **(UserID)**]
* Comment-Content[**(CommentID)**, CommentContent]
* Followship[**(FollowedID)**, **(FollowerID)**]
* Kudo[**(ActivityID)**, **(UserID)**]
* Member[**(ClubID)**, **(UserID)**]
* User[**UserID**, UserName, Gender, Age, Email]

# 

# **5. Non-Relational Data Model**

## **5.1 Choice of NoSQL Database Program**

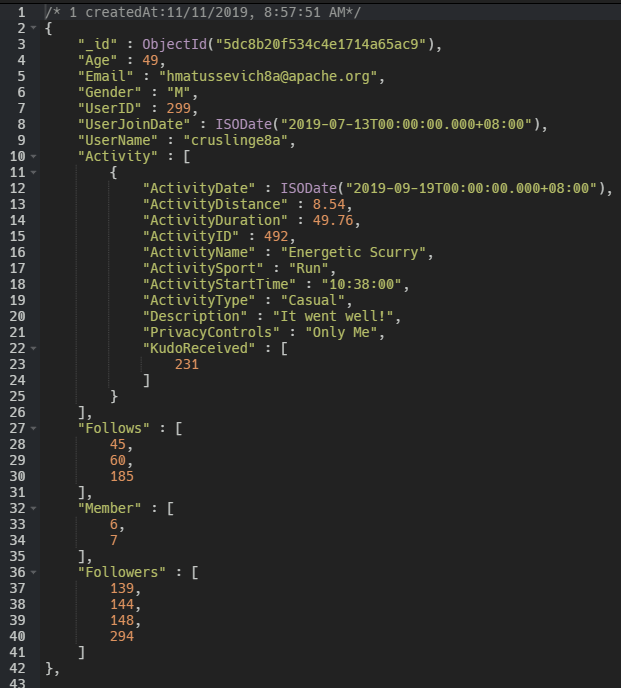
We have chosen to utilize MongoDB, a document-oriented database, for the development of our non-relational database management system. Similar to how a relational database uses rows and tables to store data, a document database like MongoDB utilizes documents and collections respectively. The main difference is that each document and collection in a MongoDB collection need not conform to a predetermined schema, and additional attributes can be added downstream to reflect and adapt to any changes in real-life data modelling needs and requirements.

In relational design, most of the initial effort is spent on creating entities and describing their relationships with other entities. The queries are usually designed after the relational schema is denormalized to minimally 3NF or BCNF. Queries with join functions are hence utilized to re-combine the data from multiple entities when performing queries. However, this is not very efficient as joins are known to cause bottlenecks on read. Meanwhile, NoSQL data modeling closely follows the concept of a “schema-free” design. The row document in the MongoDB collection would be strategically designed to allow aggregation of data that are meant to be read together.

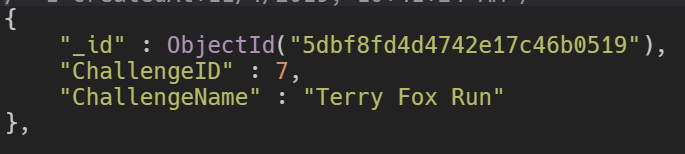
To ensure a fair comparison of performance and reliability between our relational and non-relational data models, we have chosen to reuse the synthetic application data that was generated during our relational database implementation. To facilitate the loading of these data into MongoDB, we have identified a preliminary JSON attribute-value structure for our data for each of the collections that we need in our non-relational database. According to the modelling needs of MHL, additional attribute-value fields can then be added or dropped to match these needs, thereby demonstrating the extreme flexibility that is afforded to us with a non-relational database.

## **5.2 JSON Attribute - Value Structure**

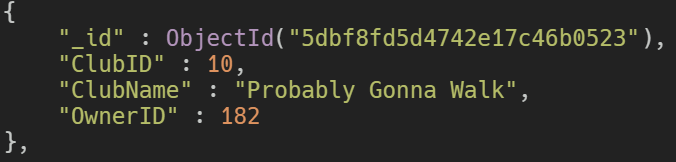
The following diagrams illustrate an example of a typical document in each of the collections that we have identified, in a JSON attribute-value-pair format.



**Fig. 1 User Collection**



**Fig. 2 Challenge Collection**



**Fig. 3 Club Collection**

### **5.2.1 User Collection**

This collection stores all personal details regarding our application users, as well as data on their activity participation, social networks (followers and following), and club memberships. The key difference between our SQL and noSQL data models lies in the fact that the latter relies on a denormalized representation of the one-to-many relationships as explained in Section 2.2.

As an example, we denormalized the one-to-many relationship between User and Activity, by embedding the various ad by a user in an array, in an attribute titled ***Activity***. With this embedded substructure approach, we can then filter collections based on the nested subfields in each element of the ***Activity*** array. In the denormalized database, we store in a single User collection, which would usually be multiple tables in a relational model representation.

The main purpose of denormalization is to remove the need for joins which can be very expensive. By embedding multiple tables in the User table, we are able to aggregate data and retrieve all of them much faster compared to using joins in SQL data model.

### **5.2.2 Challenge Collection**

This collection stores all details relevant to each challenge, which includes a challenge ID to uniquely identify each challenge, and the name of that challenge.

### **5.2.3 Club Collection**

This collection stores all details relevant to each club that was created, which includes a club ID to uniquely identify each club, as well as the name and owner of the club.

# **6. Recommendations to MHL**

Our group recommends the use of a NoSQL database model for use as backend data storage in the MHL Fitness365 mobile application. The following three considerations were taken into account in arriving at our decision.

Firstly, we anticipate the Fitness 365 mobile application to receive many future updates which involve the addition of new functionality to cater to users’ needs and preferences. With an expanding user base, MHL will very likely be expected to make constant improvements to the Fitness 365 application. With these additional functions come the need for additional data storage, new entities, and new relationships to model these changes. With a traditional SQL database, direct changes to the existing data models need to be made by database developers, and re-modelling of the various entity relationships to enforce real-world business constraints and ensure continued ACID compliance will be necessary. In a business context, this translates to additional time-to-market, which is detrimental in the field of mobile application development and technology, where speed and novelty is crucial to attracting and retaining users. In other words, by the time updates with new features are ready to be launched, they would already have become obsolete. As such, use of NoSQL database model is strongly encouraged to avoid the need for data-remodelling, and the introduction of new features can be rapid and frequent, as data storage constraints and considerations are much more lenient and flexible when it comes to non-relational data models.

Next, the database querying speed of SQL versus NoSQL databases commonly found in today’s market was compared and analyzed, and there is a general consensus that database querying is quicker, for simple queries at least, when using a NoSQL database. The MHL mobile application offers users an overview of simple statistics such as their activity history, personal best records, as well as display a list of comments and kudos received by the user in each of the activities that he/she has performed. These are all relatively simple queries, as exemplified in the 10 sample queries that our team has performed on the NoSQL database, which can be executed much faster with a NoSQL backend. Needless to say, speed is crucial when users are interacting with a mobile application. For a seamless and enjoyable user experience, applications need to be able to quickly read and write data from a backend database, either from the phone’s local storage, or from the application’s cloud servers. Delay and wait times to display and enter information and changes need to be kept to a minimum to ensure the best user experience, which is a significant factor in retaining users in MHL’s Fitness 365 mobile application. Fortunately, NoSQL serves exactly this purpose.

Lastly, we shall now examine the future scalability and feasibility for growth in using a NoSQL database. We foresee the application to enjoy steady user growth in times to come, and capacity problems will be soon be faced by standalone nodes. This is where distribution computing, and database partitioning in particular, steps in to solve these issues. NoSQL excels in providing administrators the ability to host their data on multiple nodes across different clusters, a technique commonly known as horizontal partitioning, or sharding. However, when this happens, developers need to tune their database to either favour consistency or availability, in light of the Consistency, Availability, and Partition-Tolerance (CAP) theorem. Our team recommends that consistency in data reads is not as critical as maintaining high availability to users. As an illustration, a system favouring high consistency will ensure the highest degree of coherence between the data stored on different nodes, and as a result be highly stringent in delivering and modifying data requested by end-users. This may in turn lead to frequent error messages encountered by users during read/write cycles caused by system restrictions that seek to preserve data consistency across nodes and clusters, and an unsatisfactory user experience as a result. As such, we recommend prioritising availability of data within the system to end-users, and allow minor data consistency issues to persist as the NoSQL database is scaled upwards in the future.

In conclusion, our team strongly encourages the adoption of a NoSQL database as a backend storage solution for the Fitness 365 mobile application, primarily due to its flexibility which reduces development time and time-to-market of new features, its speed and fluidity in data read requests which enhance user experience, and finally its high potential for horizontal scaling to accomodate a larger user base in the future. Nevertheless, there are certain tradeoffs associated with scaling a NoSQL database, for which our team has also provided suitable recommendations and best practices, as a guide for MHL database administrators to take into account during the development process. With these solutions and recommendations, we believe MHL is well-equipped to decide on and implement the appropriate database solution for its Fitness 365 mobile application.

# **7. Appendix**

## **7.1 Relational Database Deployment in MySQL**

1. Download the ***nationalfitnesschallenge.sql*** file in the subfolder Part 1 (SQL).
2. Open MySQL Workbench.
3. Click on the server tab at the top and choose ***[Data Import]***.
4. Under import options select***[Import from Self-Contained File]***.
5. Click on the icon with 3 dots on the right and select the ***nationalfitnesschallenge.sql*** file that was downloaded earlier.
6. Choose a default target schema if you have created one before, or simply create a new schema.
7. At the bottom, ensure the option ***[Dump Structure and Data]***is selected.
8. Lastly, go to import progress and click on ***[Start Import]***.

## 

## **7.2 Non-Relational Database Deployment in MongoDB**

1. Start MongoDB Compass and connect to the local host on your PC.
2. Start NoSQL Booster and connect the MongoDB Client. On the top-left corner, select ***Connect*** → ***Localhost***.
3. On the top of the NoSQL Booster, select the ***Import*** → ***Import from JSON and BSON files (mongoimport not required)***.
4. Click on ***Add File***, select all 3 JSON files : BC2402, Challenge, Club.
5. Ensure that the checkbox for each file is selected before clicking ***Import***.