MARCH 25, 2022

STUDENT MANAGED FUND - DATABASE

CSU 34041 - INFORMATION MANAGEMENT II ASSIGNMENT

JACK CLEARY
19333982
MSISS

TABLE OF CONTENTS

CSU 34041 - REPORT

SECTION A

1. APPLICATION DESCRIPTION

My Application is a Database Management System which stores information relating to the Student Managed Fund (SMF) Society here in Trinity College Dublin. The SMF is a long-hold equity portfolio with circa €400,000 AUM, managed entirely by a student body and governed by a professional Advisory Board.

This year, the SMF had 1,107 member signups, divided into different financial sectors and overseen by a committee. Until now, all information regarding the members, sectors, holdings etc. has all been held in several Excel Spreadsheets in quite a messy and unorganised fashion. As the newly elected Chief Technology Officer for next year, I have decided to take this opportunity to organise this information in a much more manageable and structured system.

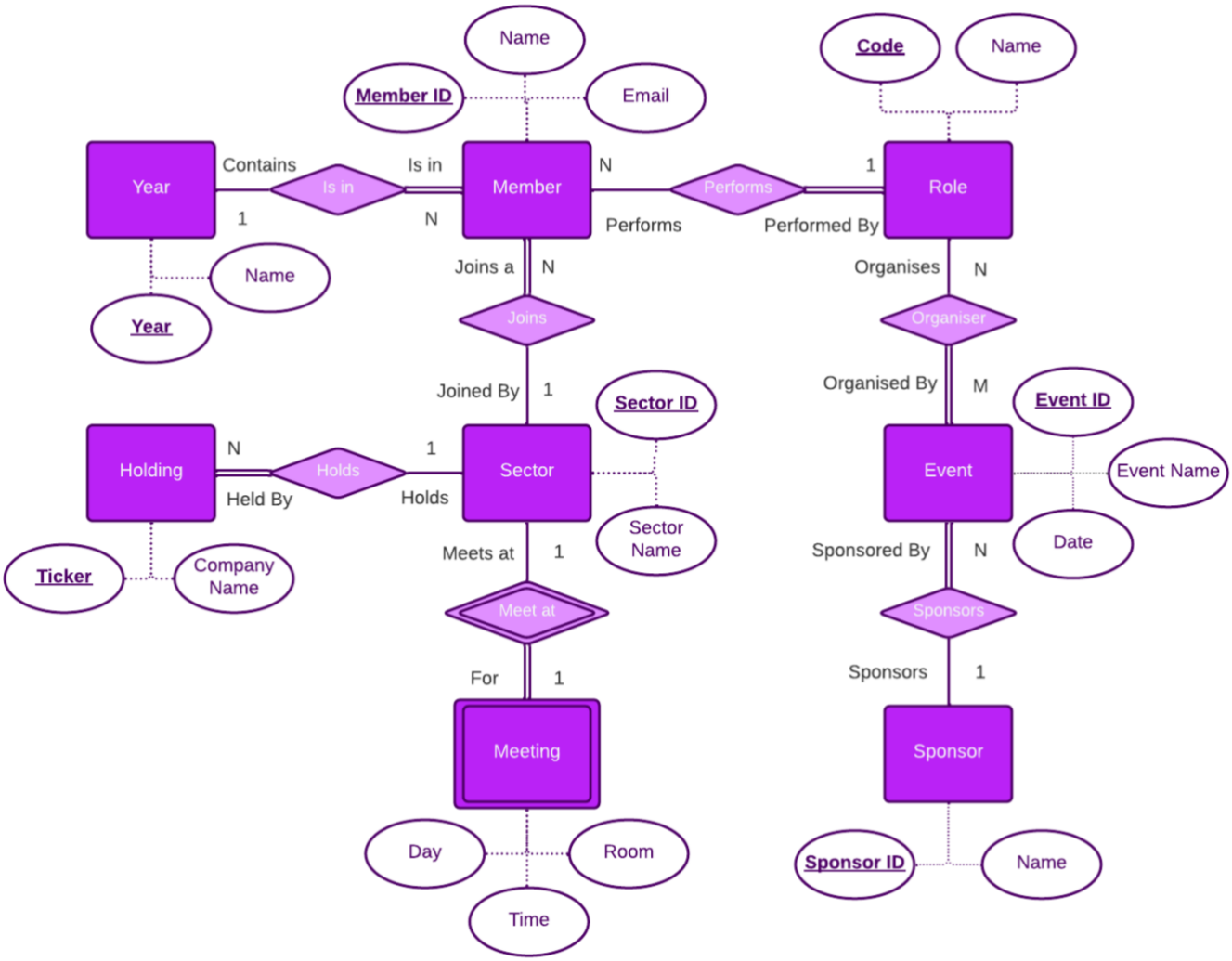
For the purpose of this assignment I will simply be designing a suitable infrastructure, but will not be populating the tables accurately, as with over 1,000 members this would probably outside the scope of the project. Hence, I will use realistic but simplified fictional data to begin.

This database will be comprised of 8 different entities, each with their own separate purpose and attributes. These are as follows:

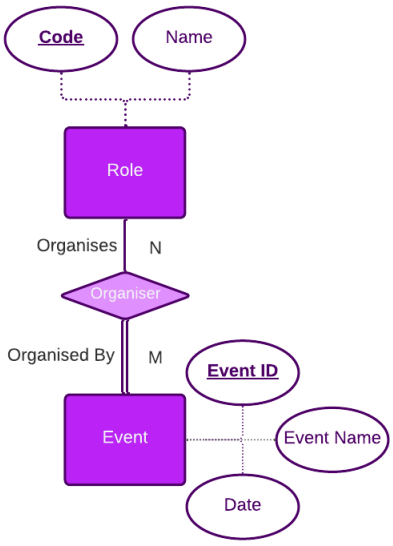
Members: Storing details of all members that join the society, name, Member ID, Year of Study, Email, Role and Sector.
Roles: A List of all the possible Roles in the SMF, from CEO down to Junior Analyst.
Sector: A list of all the different Sectors that the society is divided into, based on the type of equities they hold.
Holdings: A list of the different holdings that the SMF own, along with the sector that they relate to.
Meetings: A list of details of the Meetings of the different Sectors, day, time and location.
Years: A list of the different years by number, i.e. Junior Fresh up to Senior Sophister and Other.
Sponsors: A list of the very generous companies who sponsor the SMF, both by sponsoring events and donating to the fund itself.
Events: A list of all of the events run by the SMF, their ID, Name, Date and Sponsor ID.

My aim was to develop a very user friendly and simple database which can be used and built on for years to come. As I stated previously, at the moment all information relating to members, events, sponsors etc. is sitting in various Excel spreadsheets and in my opinion this is not a very efficient way to manage the data. This assignment has given me the opportunity to build something very practical and useful, and for that I will actually say thank you!

2. ENTITY RELATIONSHIP DIAGRAM



Shown above is the Entity Relationship Diagram for my DBMS. The entities are represented by the dark purple squares, relationships by the lighter purple diamonds and attributes are found in the white ovals.

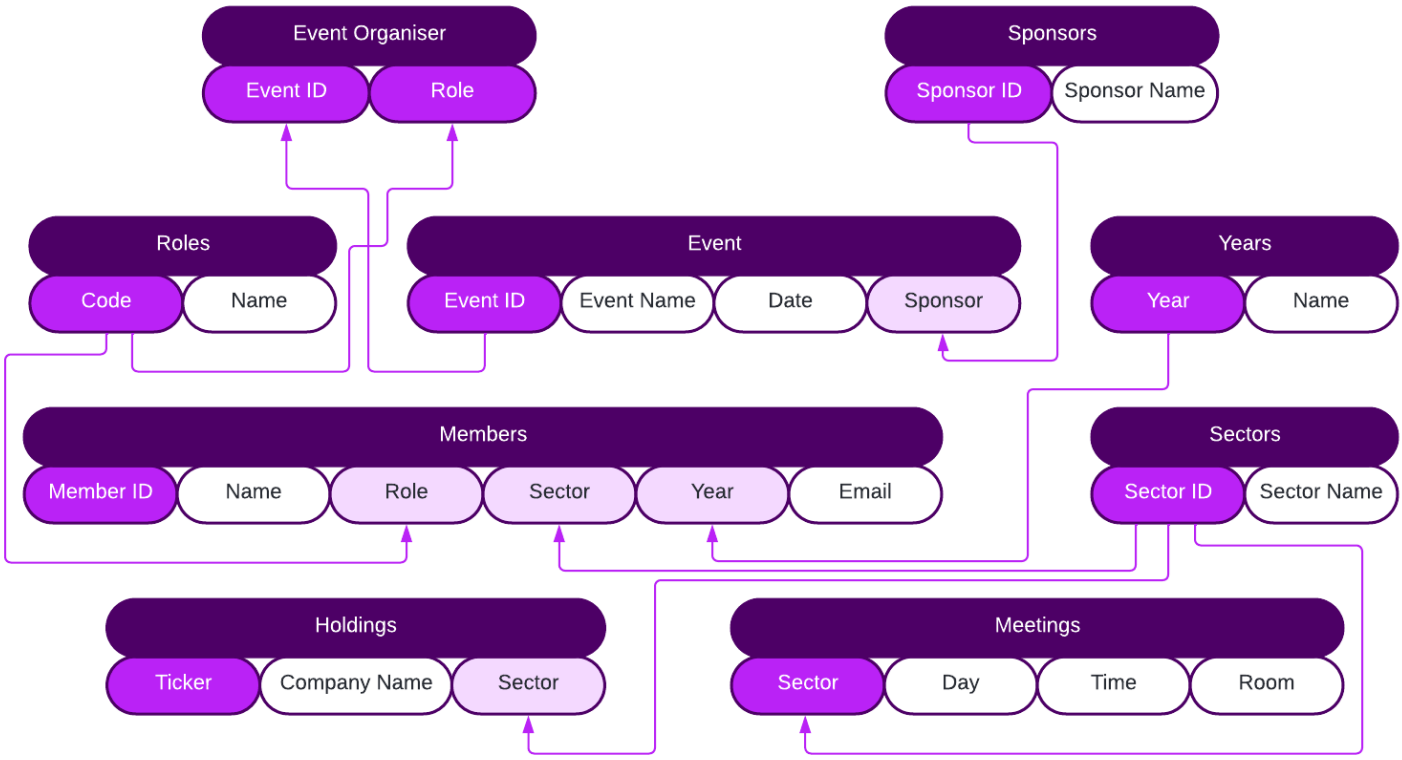


As seen below, the primary keys are written in bold and underlined. Foreign keys are not represented on the ER Diagram.

The single lines between entities and relationships indicate partial participation, whereas the double lines indicate total participation. To the left we see an example of this, in the relationship between Sponsors and Events held by the society. In this instance, every Event must have a sponsor in order for it to be paid for, however not every sponsor will necessarily pay for an event, with some just donating money to the fun itself.

The 1 and N indicate the 1:N relationship between these two entities, in that every event will have only one sponsor, but one company can sponsor many events. Of the 7 total relations in the database, 5 are 1:N, one is 1:1 and the last is N:M.

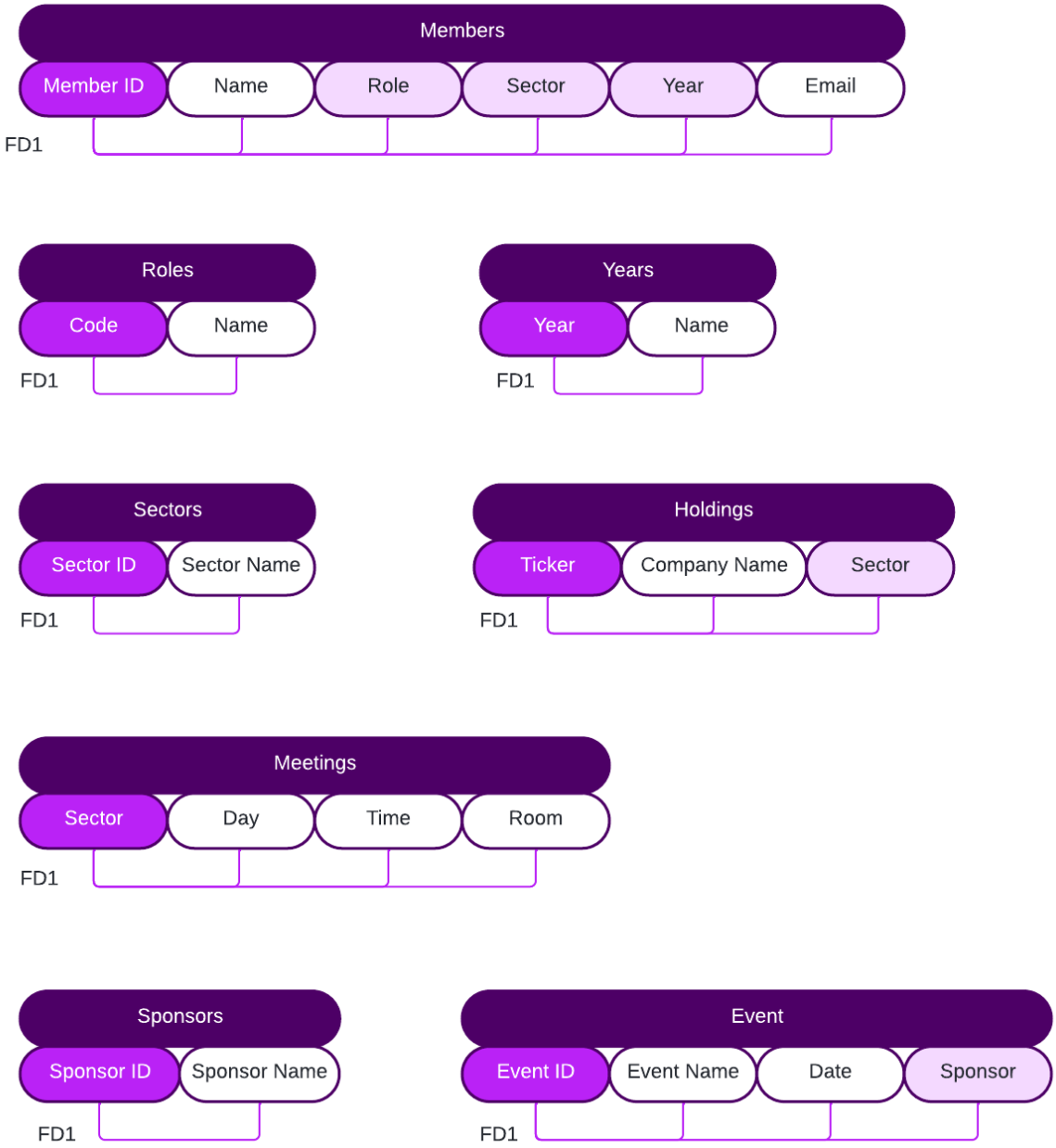
3. MAPPING TO RELATIONAL SCHEMA



Above is the Relational Schema Diagram, which demonstrates the relationships between the attributes in each of the tables. The primary keys are coloured in the darker purple, and the foreign keys are the paler purple.

The N:M relationship between the Roles and Events entities required the creation of the Event Organiser table. This table contains a composite primary key, comprised of the Event ID and the Role Code. This is the case because each organiser can do several events and each event can also have several organisers.

4. FUNCTIONAL DEPENDENCY DIAGRAMS



Above is the Functional Dependency Diagram, which demonstrates any dependencies which may exist within each table. For example, in the members table all of the attributes are determined by the Member ID and so we can say that all of the other attributes in this table are functionally dependent on the Member ID.

Each table is also in the Third Normal Form, as at each row/column intersection there is only one value (satisfying the 1NF conditions), each non-key attribute is functionally dependent on the primary key (satisfying the 2NF conditions), and lastly no key is transitively dependent on the primary key.

SECTION B

5. CREATING DATABASE TABLES

Creating tables in MYSQL is a very straightforward process thanks to the many useful statements and keywords available for use. For this database, I created 9 tables, each with their own different keys, attributes and constraints.

I employed the following steps when creating a table: Firstly, choose a type for each of the attributes in the table. Next, choose a primary key for the table, and ensure it is a key that will be unique in each row. My next step was to decide what constraints to place on each of the attributes in the table, i.e. whether or not they can be NULL, whether they should be unique or if they should be in a certain format etc. I then picked out any foreign keys from the table and defined them as such, ensuring that they were referencing the correct attributes. Lastly, I would think about if there were any other constraints that should be placed on the table or on new entries, for example is there a limit on the number of tuples in the table etc.

```
CREATE TABLE `Members` (
  `Member ID` int(4) NOT NULL AUTO_INCREMENT,
  `Name` varchar(30) NOT NULL,
  `Role` varchar(20) NOT NULL ,
  `Sector`int(2) NOT NULL ,
  `Year` int(1) NOT NULL ,
  FOREIGN KEY `fk_Role` (`Role`) REFERENCES Roles(`Code`) ON DELETE CASCADE ON UPDATE CASCADE,
  FOREIGN KEY `fk_Sector` (`Sector`) REFERENCES Sectors(`Sector ID`) ON DELETE CASCADE ON UPDATE CASCADE,
  FOREIGN KEY `fk_Year` (`Year`) REFERENCES Years(`Year`) ON DELETE CASCADE ON UPDATE CASCADE,
  PRIMARY KEY (`Member ID`)
);
```

Above is the SQL code I used to create the Members table for the database. The attributes listed (member ID, Name, Role, Sector and Year) are all given the NOT NULL constraint, meaning that they are required entries and cannot be left as NULL. I initialised the Member ID, Sector and Year variables as an integer of different lengths, and the Name and Role variables as variable character fields.

The Member ID is also initialised with the AUTO_INCREMENT keyword, meaning that every time there is a new input in the table the Member ID will automatically increment by 1.

Role, Sector and Year are all Foreign Keys and hence are created as such, with each referencing the table in which it is a primary key. In each case the ON DELETE CASCADE and ON UPDATE CASCADE options are used, so for example if the name of a role is updated in the Roles table, the value in the table will change accordingly, and if a sector is disbanded or removed from the Sectors table then any member in that sector will also be deleted from the database. I contemplated using ON DELETE RESTRICT here but ultimately decided that if a sector were to be deleted it would usually mean that those members could not be reallocated to different sectors anyway and so would no longer be members.

Lastly, the Member ID is defined as the primary key for this table. In creating this table, I initially forgot to add the Email attribute, however this problem could be solved later using the ALTER TABLE statement.

6. ALTERING TABLES

The ALTER TABLE statement can be used to edit a table that you have already created, by adding or removing columns, constraints and any other features that may need to added or removed. This function is very useful for when mistakes are spotted in the design of a database, and also for optimizing the database's efficiency.

As mentioned briefly above, I had forgotten to include the Email attribute in my Members table in this database. To correct this error I employed the use of the ALTER TABLE statement, as seen below.

```
ALTER TABLE `Members` ADD `Email` varchar(50) NOT NULL UNIQUE AFTER `Year`;
ALTER TABLE `Members` ADD CONSTRAINT CHECK (`Email` LIKE '%@tcd.ie');
```

The first line of code adds the Email column to the table, defining it as a variable character field with maximum length of 50 characters, and insisting that it should not be NULL and that each email in the system should be unique. This would avoid duplicate entries into the database, and lastly the column is inserted after the Year column already in the table.

The second line of code adds a constraint to this table, stating that any email entered must end in '@tcd.ie', which would restrict the members to use of their Trinity emails alone and ensures no fake emails are added to the list.

7. TRIGGER OPERATIONS

In SQL, Triggers are a stored program which are invoked automatically when a particular specified event occurs, for example insertion of data into a table, updating or deletion. This allows you to automatically edit a table or other tables when data is inputted. In this SMF database, there was not much of a requirement for triggers, so I only used one. Below is the code for this trigger:

```
delimiter //
CREATE TRIGGER `Verify Event Date` BEFORE INSERT
ON `Events`
FOR EACH ROW
IF NEW.`Date` >= curdate() THEN
SIGNAL SQLSTATE '50001' SET MESSAGE_TEXT = "Input Date is invalid, cannot be a future date.";
END IF;
//
```

As we can see, I am verifying the dates entered into the events table, i.e. ensuring that dates entered are not a future date. This is because we only want the database to record events that have already happened, not future prospects. I originally tried to use a CHECK constraint to verify these dates, however due to the fact that the 'curdate()' function needs to be run every time data is inputted, it was necessary to implement a trigger instead.

As stated above, there was not any other requirement for triggers in this database, as most of the data that is stored is stand alone and does not affect any other tables. The Foreign keys in each table are already constrained by their nature and so it was not necessary to implement triggers regarding any of these attributes.

8. CREATION OF VIEWS

Views are another extremely useful aspect of databases in SQL. These allow for quick and clear representation of data in any specified format. They also allow for tables to be combined and for certain information to be withheld, which can be beneficial from a security point of view. I defined 7 different views for the purpose of this project, firstly showing all of the members in each of the different sectors, and also showing counts of the number of members in each sector and the number of holdings in each sector. These views would be beneficial as it would provide individual Sector Managers with details of members in their respective sectors without having to give them access to the entire membership list. This is ideal from a privacy point of view, but also with respect to practicality, as Sector Managers won't have to sift through a list of over 1,000 members and can focus solely on their own sector. Below is the code used to create one of said views:

```
DROP VIEW IF EXISTS `Energy Sector`;
CREATE VIEW `Energy Sector`
AS
SELECT `Member ID`, `Name`, `Role`, `Year`, `Email` FROM `Members` WHERE `Sector` = 2;
```

This code very simply selects all of the instances in the Members table where the Sector attribute is equal to 2. This number then corresponds to the Energy Sector, and hence it takes all columns except for the Sector column (now deemed irrelevant) and effectively creates a new table with the information of all members in the Energy Sector.

As mentioned above, I also created a view to count the number of members in each sector, and this code which does this can be seen here:

```
DROP VIEW IF EXISTS `Number of Members by Sector`;
CREATE VIEW `Number of Members by Sector`
AS
SELECT `Sectors`.`Sector Name`, COUNT(*) AS `Number of Members`
FROM `Members`
INNER JOIN `Sectors` ON `Sectors`.`Sector ID` = `Members`.`Sector`
GROUP BY `Sector`;
```

This code was slightly more complicated, as it required selecting columns from more than one table and joining them using the INNER JOIN command. This allowed me to join columns from the Members table and the Sectors table, in order to provide the view seen on the right. The columns are joined on the Sector IDs meaning rows are matched up by having equal Sector IDs. The information in the Members table is also grouped by Sector and hence there were equal numbers of rows in each table.

Sector Name	Number of Members
Committee	6
Energy	2
Discretionary	3
Technology - Softare	2

This information will be beneficial to the committee as it will indicate if certain sectors are over/under weighted with members and hence aid in decision making of where to allocate new sign ups.

9. POPULATING TABLES

Populating the tables is fairly straightforward, and relies on the fact that all necessary constraints are in place and working effectively. The code below was used to populate the events table, which keeps track of any events that the SMF have held during the year.

```
INSERT INTO `Events` VALUES ("LPC", "Leadership Perspectives Conference", "2022-03-03", "1");
INSERT INTO `Events` VALUES ("WIB", "Women in Business Conference", "2022-02-05", "3");
INSERT INTO `Events` VALUES ("GS", "Guest Speaker", "2021-11-28", "5");
INSERT INTO `Events` VALUES ("EDU", "Educlass", "2021-12-03", "4");
INSERT INTO `Events` VALUES ("AIC", "Alternative Investments Conference", "2021-10-26", "2");
```

Each line fairly straightforwardly inserts a tuple into the table, containing each of the necessary attributes (Event ID, Event Name, Date and Sponsor). These entries are checked against the constraints that I have put in place, and in this instance they are also verified using the trigger mentioned previously. As we can see, each of these dates are in the past, and hence they are all valid date entries. As well as this, Sponsor ID is a foreign key from the Sponsor table, and each of these values match an entry in this parent table, so hence all constraints are satisfied and the table can be populated.

10. RETRIEVING INFORMATION FROM THE DATABASE

Retrieving information from the database can be done using the SELECT clause in conjunction series of other commands. This is a very useful tool, particularly for the SMF database, due to the fact that most information will be inputted at the beginning of the year and for the rest of the year it will mainly be used to display the data in any way necessary.

Below is an example of a simple query I used to list each event along with the name of the sponsor of the event. This required the use of the INNER JOIN statement, to join the Events and Sponsors tables, on the Sponsor ID. However, the Sponsor ID itself is not overly relevant in this case, so I only use the Sponsor Name, Event Name and Event Date columns.

```
SELECT `Sponsors`.`Name` AS `Event Sponsor`, `Events`.`Event Name`, `Events`.`Date`
FROM `Sponsors`
INNER JOIN `Events` ON `Events`.`Sponsor` = `Sponsors`.`Sponsor ID`;
```

For simplicity, I have used Views to retrieve the data in the various forms I imagined it being required, and hence I have rarely used select and join statements on their own. Below is another example of a join, which shows the number of holdings each sector has.

Again this uses the INNER JOIN and concatenates 2 columns, the Sector name and the newly created count of the number of holdings in each sector. This is quite similar to the number of members view above in Section 8.

```
DROP VIEW IF EXISTS `Number of Holdings by Sector`;
CREATE VIEW `Number of Holdings by Sector`
AS
SELECT `Sectors`.`Sector Name`, COUNT(*) AS `Number of Holdings`
FROM `Holdings`
INNER JOIN `Sectors` ON `Sectors`.`Sector ID` = `Holdings`.`Sector`
GROUP BY `Sector`;
```

11. SECURITY COMMANDS

Security and Access Control are very important for a database as they will prevent unauthorised persons from accessing the system in order to obtain information or make malicious changes.

In order to implement security measures for my database, I used a Role-Based Access system. I created two different roles, one which offers complete access and would be granted to users such as the CEO and CTO, and another for the rest of the committee members who do not require quite the same privileges. Below is the code I used to create the committee member roles, and then create users for each committee member and assign them each this role.

```
DROP ROLE IF EXISTS committee_member@'%';
CREATE ROLE committee_member@'%';
GRANT SELECT ON smf_db.* TO committee_member;
GRANT INSERT,DELETE,UPDATE ON smf_db.`Members` TO committee_member;

DROP USER IF EXISTS Lucy@'%';
CREATE USER Lucy@'%' IDENTIFIED BY "smf123";
GRANT committee_member TO Lucy@'%';

DROP USER IF EXISTS Jane@'%';
CREATE USER Jane@'%' IDENTIFIED BY "smf123";
GRANT committee_member TO Jane@'%';

DROP USER IF EXISTS Aoife@'%';
CREATE USER Aoife@'%' IDENTIFIED BY "smf123";
GRANT committee_member TO Aoife@'%';
```

As we can see, these users will be granted the ability to view all tables in the database and query all of the information, however they will only be able to insert, delete and update data in the Members

table. This will allow for better security and is also more practical, as these committee members will never need to alter any other information in the database.

12. ADDITIONAL FEATURES

Some additional features which I decided to make use of were functions and stored procedures. As mentioned above, the SMF have several very generous sponsors, and these sponsors can have different sponsorship levels based on the number of events that they run. The different levels are Platinum, Gold and Silver. In order to indicate which sponsorship level each has, I used a function which checks the number of events a company has sponsored, and this function is then called within the stored procedure called "getSponsorLevels". Below is the code I used to create both the function and the procedure:

```
DELIMITER $$
CREATE FUNCTION SponsorLevel(
    no_events int
)
RETURNS VARCHAR(8)
DETERMINISTIC
BEGIN
    DECLARE sponsorLevel VARCHAR(8);
    IF no_events > 2 THEN
        SET sponsorLevel = 'PLATINUM';
    ELSEIF no_events = 2 THEN
        SET sponsorLevel = 'GOLD';
    ELSEIF no_events < 2 THEN
        SET sponsorLevel = 'SILVER';
    END IF;
    RETURN (sponsorLevel);
END$$
DELIMITER ;
```

```
DELIMITER //
CREATE PROCEDURE getSponsorLevels()
BEGIN
    SELECT `Sponsor ID`, `Name`, SponsorLevel(`Number of Events`) AS `Sponsor Level`
    FROM smf_db.`number of events by sponsor`;
END //
DELIMITER ;
```

This will allow the COO, who is in constant communication with sponsors and will be given full access to the database, to quickly ascertain the sponsorship level of the different companies virtually at the push of a button.