

CODDs Rules

RULE 1: THE INFORMATION RULE

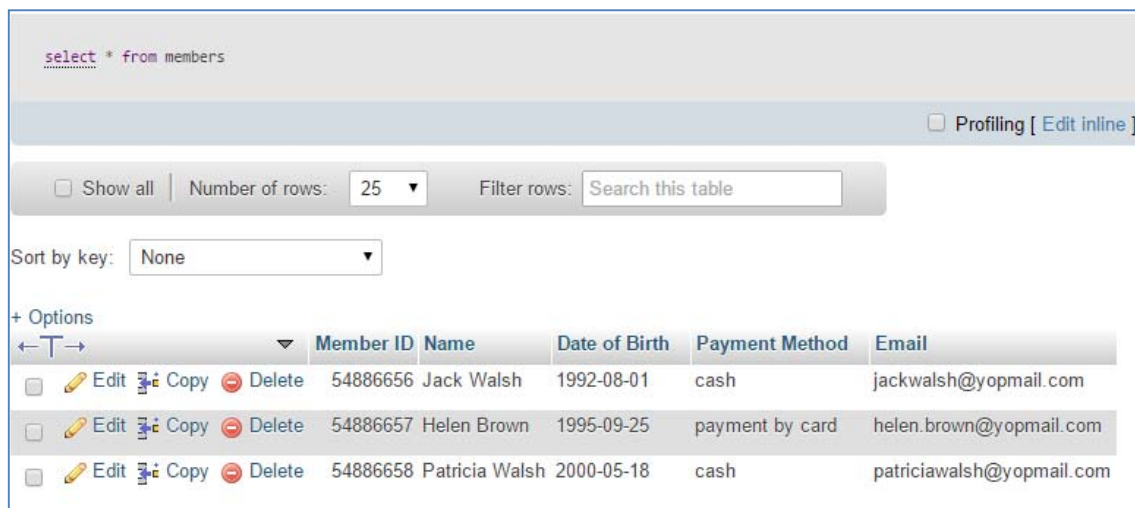
All information should be stored in a table form.

Evidence of satisfaction of rule 1:

This rule is satisfied by the fact that I am using a relational model to represent all the data of my project. The data is presented as multiple tables. The data from each table can be shown to the user without any imposed order. An example of data can be seen after executing the following query:

Query:

Select* from members;



The screenshot shows a database query result interface. At the top, the query `select * from members` is entered. Below the query, there are controls for displaying the results: a checkbox for 'Show all', a 'Number of rows' dropdown set to 25, and a 'Filter rows' search box. A 'Sort by key' dropdown is set to 'None'. A '+ Options' link is visible. The table data is displayed with columns: Member ID, Name, Date of Birth, Payment Method, and Email. Each row has interactive icons for Edit, Copy, and Delete.

	Member ID	Name	Date of Birth	Payment Method	Email
<input type="checkbox"/> Edit Copy Delete	54886656	Jack Walsh	1992-08-01	cash	jackwalsh@yopmail.com
<input type="checkbox"/> Edit Copy Delete	54886657	Helen Brown	1995-09-25	payment by card	helen.brown@yopmail.com
<input type="checkbox"/> Edit Copy Delete	54886658	Patricia Walsh	2000-05-18	cash	patriciawalsh@yopmail.com

RULE 2: GUARANTEED ACCESS RULE

Data should be accessed without ambiguity by the combination of:

Table name + Primary Key (Row) + Attribute (Column).

The table name locates the correct table, the column name finds the correct column, and the primary key value finds the row containing an individual data item of interest.

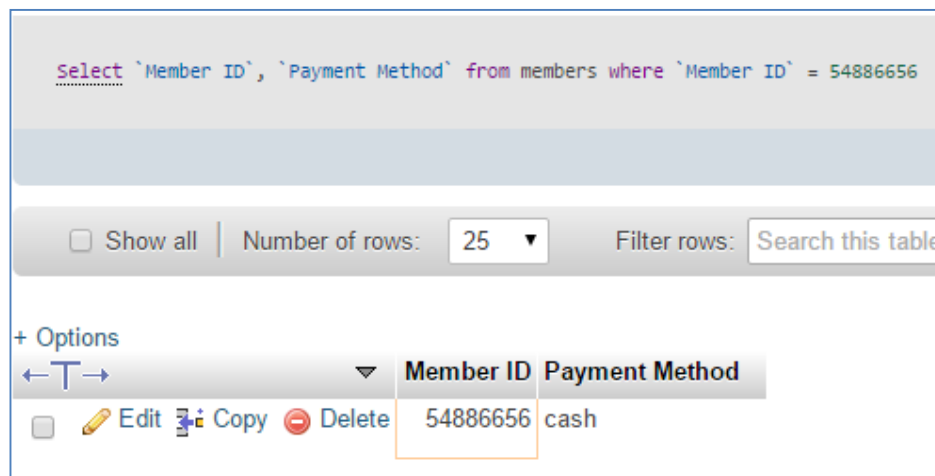
Primary keys ensure uniqueness of every row and avoid ambiguities when searching for data.

Evidence of satisfaction of rule 2:

This rule is satisfied by the fact that all the tables that I have created in this project include primary keys. The use of primary keys guarantees the selection of a unique data entry in a particular table. In the following, I am giving two examples of queries showing ambiguous and unambiguous results using respectively a query with a primary key and without a primary key.

Unambiguous Query: where 'member id' is the primary key:

Select `Member ID`, `Payment Method` from members where `Member ID` = 54886656;



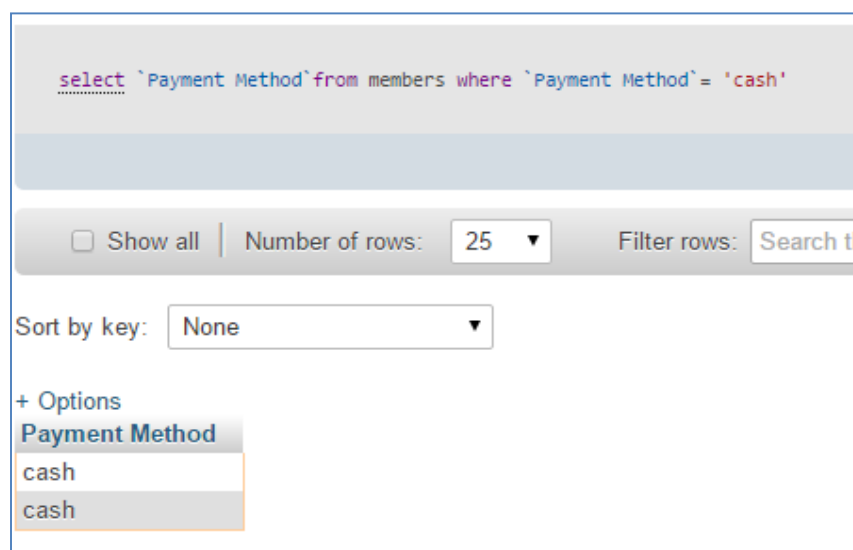
The screenshot shows a database query interface. At the top, the SQL query is: `Select `Member ID`, `Payment Method` from members where `Member ID` = 54886656`. Below the query, there are controls for 'Show all', 'Number of rows' (set to 25), and a 'Filter rows' search box. Underneath, there are '+ Options', a table icon, and a dropdown arrow. The table has two columns: 'Member ID' and 'Payment Method'. A single row is displayed with the values '54886656' and 'cash'. Below the table, there are icons for 'Edit', 'Copy', and 'Delete'.

Member ID	Payment Method
54886656	cash

If we only select column name, this can create ambiguity as many members can have the same payment method:

Ambiguous Query: where there is no primary key:

select `Payment Method` from members where `Payment Method` = 'cash';



The screenshot shows a database query interface. At the top, the SQL query is: `select `Payment Method` from members where `Payment Method` = 'cash'`. Below the query, there are controls for 'Show all', 'Number of rows' (set to 25), and a 'Filter rows' search box. Underneath, there is a 'Sort by key' dropdown set to 'None'. Below that, there are '+ Options' and a table icon. The table has one column: 'Payment Method'. Two rows are displayed, both with the value 'cash'.

Payment Method
cash
cash

RULE 3: SYSTEMATIC TREATMENT OF NULL VALUES

a. Missing data is represented by 'NULL' value:

When there is no information, there should be no blank space and no zero; there should be a 'NULL' value instead.

For this to happen, the default value should be 'NULL' for all attributes that may not contain any information.

Evidence of satisfaction of rule 3.a:

In order to show that this feature is possible, I refer to the following figure that shows that all attributes except for the primary key (i.e., Payment Reference) have as default value 'NULL' in case the attribute is not specified.

Structure						
Name	Type	Length/Values	Default	Collation	Attributes	Null A_I C
Payment Reference	VARCHAR	8	None	latin1_swedish_c		<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>
Member ID	INT	8	NULL			<input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>
Club ID	VARCHAR	4	NULL	latin1_swedish_c		<input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>
Payment Method	VARCHAR	20	NULL	latin1_swedish_c		<input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>
Amount	INT	5	NULL			<input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>
Payment Date	DATE		NULL			<input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>
Annual Membership i	DATE		NULL			<input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>

In order to test and validate this, I use the following query to insert an entry to this table with missing attributes (i.e., 'Member ID' and 'Annual Membership Renewal Date').

Query:

```
insert into payments( `Payment Reference`, `Club ID`, `Payment Method`, `Amount`,  
`Payment Date`) values ('P0020291', 'C015', 'payment by card', 500, 2015-06-30);
```

✓ 1 row inserted. (Query took 0.0909 seconds.)

```
insert into payments( `Payment Reference`, `Club ID`, `Payment Method`, `Amount`, `Payment Date`) values ('P0020291', 'C015', 'payment by card', 500, 2015-06-30)
```

After the insert query, I refer to the following table that shows NULL for attributes without values (this is neither zero nor an empty string).

+ Options								
		Payment Reference	Member ID	Club ID	Payment Method	Amount	Payment Date	Annual Membership Renewal Date
<input type="checkbox"/>	Edit Copy Delete	P0020288	54886657	NULL	payment by card	200	2015-01-31	2016-01-31
<input type="checkbox"/>	Edit Copy Delete	P0020289	54886658	NULL	cash	200	2015-01-03	2016-01-03
<input type="checkbox"/>	Edit Copy Delete	P0020290	54886656	NULL	Cash	200	2015-02-03	2016-02-03
<input type="checkbox"/>	Edit Copy Delete	P0020291	NULL	C015	payment by card	500	2015-06-30	NULL

b. No 'NULL' values for primary keys:

Primary keys must not have 'NULL' values as they should never be empty.

Evidence of satisfaction of rule 3.b:

In order to show that this part of the rule is covered, I tried to enter data without a value to its primary key using the following query:

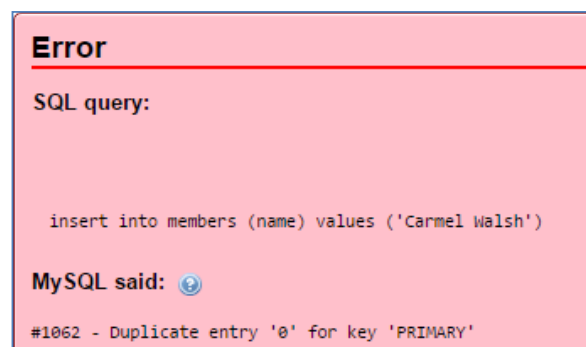
Query:

Insert into members (name) values ('Jessica Simpson');

The result of this query is shown in the following table. Notice that the first line contains the entry with name 'Jessica Simpson' with null values in all the other attributes except for the 'Member ID'. The 'Member ID' has been assigned the value 0 which is different from NULL.

	Member ID	Name	Date of Birth	Payment Method	Email
<input type="checkbox"/>	0	Jessica Simpson	NULL	NULL	NULL
<input type="checkbox"/>	54886656	Jack Walsh	1992-08-01	cash	jackwalsh@yopmail.com
<input type="checkbox"/>	54886657	Helen Brown	1995-09-25	payment by card	helen.brown@yopmail.com
<input type="checkbox"/>	54886658	Patricia Walsh	2000-05-18	cash	patriciawalsh@yopmail.com

To further verify this feature, I am trying to insert another entry with no value in the 'Member ID'. The system does not allow me to do so as it uses 0 as a default value that already exists. This is shown in the following figure.



RULE 4: DYNAMIC ON-LINE CATALOG BASED ON THE RELATIONAL MODEL

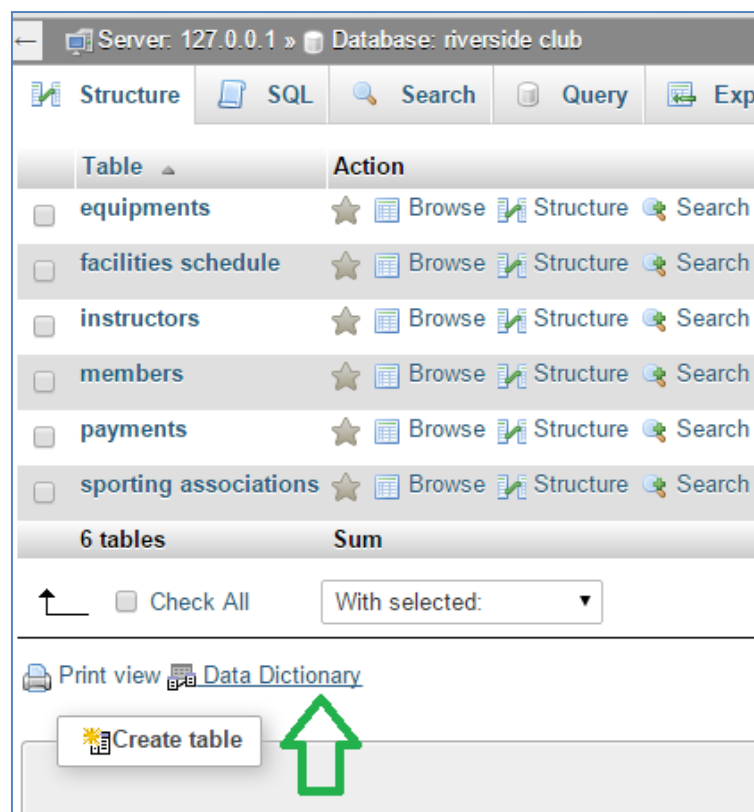
This rule requires that a relational database be self-describing. In other words, the database must contain certain system tables whose columns describe the structure of the database itself.

A relational database must provide access to its structure through the same tools that are used to access the data.

Evidence of satisfaction of rule 4:

Rule 4 is satisfied by the fact that I am using phpMyAdmin for managing my database starting from its creation to the entry of data.

This tool gives access at the same time to the data structure as well as the data itself. The structure of database, called also the Data Dictionary can be accessed by clicking on the 'Data Dictionary' link in phpMyAdmin as shown in the following figure.



After clicking this link, the data dictionary of my database is as follows.

equipments

Column	Type	Null	Default	Links to	Comments
Equipment Code (<i>Primary</i>)	varchar(7)	No			
Equipment Name	varchar(30)	Yes	NULL		
Inspection Date (<i>Primary</i>)	date	No			
Inspection Time (<i>Primary</i>)	time	No			
Instructor ID	varchar(5)	Yes	NULL	instructors -> Instructor ID	
Report	text	Yes	NULL		
Report Number	varchar(7)	Yes	NULL		
Repair Status	varchar(10)	Yes	NULL		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
PRIMARY	BTREE	Yes	No	Equipment Code	1	A	No	
				Inspection Date	1	A	No	
				Inspection Time	1	A	No	
Instructor ID	BTREE	No	No	Instructor ID	1	A	Yes	

facilities schedule

Column	Type	Null	Default	Links to	Comments
Facility Name (<i>Primary</i>)	varchar(30)	No			
Date (<i>Primary</i>)	date	No			
Start Time (<i>Primary</i>)	time	No			
End Time (<i>Primary</i>)	time	No			
Member ID	int(8)	Yes	NULL	members -> Member ID	
Club ID	varchar(4)	Yes	NULL	sporting associations -> Club ID	

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
PRIMARY	BTREE	Yes	No	Facility Name	3	A	No	
				Date	3	A	No	
				Start Time	3	A	No	
				End Time	3	A	No	
Member ID	BTREE	No	No	Member ID	3	A	Yes	
Club ID	BTREE	No	No	Club ID	3	A	Yes	

instructors

Column	Type	Null	Default	Links to	Comments
Booking Reference (<i>Primary</i>)	varchar(10)	No			
Instructor ID (<i>Primary</i>)	varchar(5)	No			
Instructor Name	varchar(30)	Yes	NULL		
Facility Name	varchar(30)	Yes	NULL	facilities schedule -> Facility Name	
Member ID	int(8)	Yes	NULL	members -> Member ID	
Date	date	Yes	NULL	facilities schedule -> Date	
Start Time	time	Yes	NULL	facilities schedule -> Start Time	
End Time	time	Yes	NULL	facilities schedule -> End Time	

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
PRIMARY	BTREE	Yes	No	Booking Reference	0	A	No	
				Instructor ID	0	A	No	
Facility Name	BTREE	No	No	Facility Name	0	A	Yes	
Member ID	BTREE	No	No	Member ID	0	A	Yes	
Date	BTREE	No	No	Date	0	A	Yes	
Start Time	BTREE	No	No	Start Time	0	A	Yes	
End Time	BTREE	No	No	End Time	0	A	Yes	
Foreign Key fk6	BTREE	No	No	Facility Name	0	A	Yes	
				Date	0	A	Yes	
				Start Time	0	A	Yes	
				End Time	0	A	Yes	

members

Column	Type	Null	Default	Links to	Comments
Member ID (<i>Primary</i>)	int(8)	No			
Name	varchar(30)	Yes	NULL		
Date of Birth	date	Yes	NULL		
Payment Method	varchar(20)	Yes	NULL		
Email	varchar(30)	Yes	NULL		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
PRIMARY	BTREE	Yes	No	Member ID	3	A	No	

payments

Column	Type	Null	Default	Links to	Comments
Payment Reference (<i>Primary</i>)	varchar(8)	No			
Member ID	int(8)	Yes	NULL	members -> Member ID	
Club ID	varchar(4)	Yes	NULL	sporting associations -> Club ID	
Payment Method	varchar(20)	Yes	NULL		
Amount	int(5)	Yes	NULL		
Payment Date	date	Yes	NULL		
Annual Membership Renewal Date	date	Yes	NULL		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
PRIMARY	BTREE	Yes	No	Payment Reference	4	A	No	
Member ID	BTREE	No	No	Member ID	4	A	Yes	
Club ID	BTREE	No	No	Club ID	4	A	Yes	

sporting associations

Column	Type	Null	Default	Links to	Comments
Club ID (<i>Primary</i>)	varchar(4)	No			
Club Name	varchar(30)	Yes	NULL		
Contact Details	int(10)	Yes	NULL		

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
PRIMARY	BTREE	Yes	No	Club ID	1	A	No	

Tables can also be accessed using the following query:

Query:

show tables;

Other queries can also be used such as :

select table_name from information_schema.tables where table_schema='riverside club'

A user can further visualise the description of a table using the command describe such :

describe equipments;

(where 'equipments' is a table in my database)

RULE 5: COMPREHENSIVE DATA SUBLANGUAGE RULE

The language to use by the DBMS need to handle creating of table structures, views, manipulate data of tables, handle integrity constraints as well as perform transactional processing operations.

Evidence of satisfaction of rule 5:

In this project I use MySQL that uses the Structure Query Language (SQL). Even though Codd does not consider SQL as fully satisfying this rule, I notice that MySQL satisfies it. Indeed, using MySQL, it is possible to:

- create tables → data definitions

Query:

**create table `sporting associations` (`Club ID` varchar(4) NOT NULL DEFAULT ,
`Club Name` varchar(30) DEFAULT NULL, `Contact Details` int(10) DEFAULT NULL);**

- create views → view definitions

Query:

create view details as SELECT `Member ID`, `Name` from members;

- Insert, delete and update data → data manipulation

Query:

INSERT into members values(54886656, 'Jack Walsh', '1992-08-01', 'cash', 'jackwalsh@yopmail.com');

- Primary keys, not null values, foreign keys, etc. → integrity constraints

Query:

```
ALTER TABLE `members` ADD PRIMARY KEY (`Member ID`);
```

- Commit, rollback operations → Transaction boundaries

Query:

```
commit;
```

RULE 6: VIEW UPDATING RULE

The views that are theoretically updatable are also updatable by the system.

Evidence of satisfaction of rule 6:

To satisfy this rule, DBMS provider should allow to propagate changes on views to the original tables. To verify this feature, I test changes on views create on a single then on multiple tables.

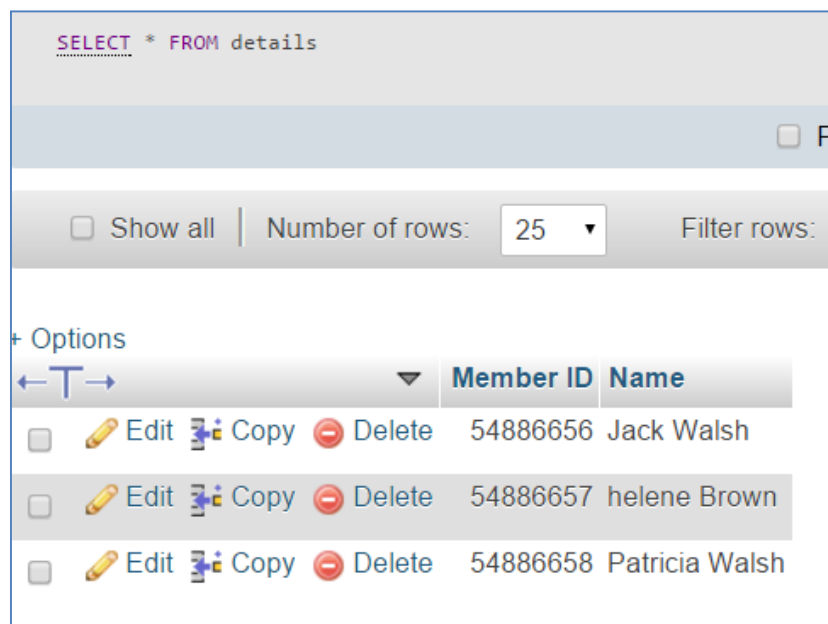
a. View on a single table:

The query below shows the command I used to update a view:

Query:

```
update details set `Name`= 'helene Brown' where `Name`= 'helen Brown';
```

The figure below shows how the view was updated:

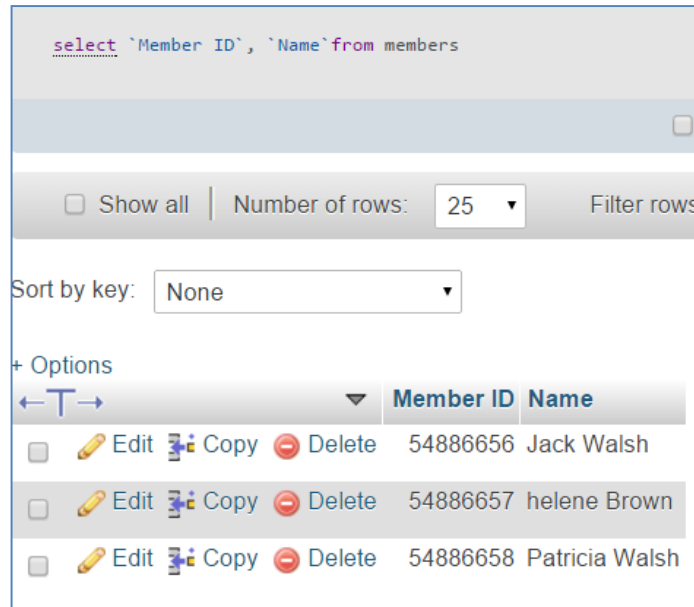


SELECT * FROM details	
Member ID	Name
54886656	Jack Walsh
54886657	helene Brown
54886658	Patricia Walsh

To verify that this change has been propagated to the original table, I execute the following select query and the results are shown in the following figure. The results show that the original table has been updated.

Query:

select `Member ID`, `Name` from members;



	Member ID	Name
<input type="checkbox"/> Edit Copy Delete	54886656	Jack Walsh
<input type="checkbox"/> Edit Copy Delete	54886657	helene Brown
<input type="checkbox"/> Edit Copy Delete	54886658	Patricia Walsh

b. View on 2 tables:

I create a view that includes data from two tables:

Query:

**create view view1 as select members.`Member ID`, `Amount` from members, payments
where members.`Member ID` = payments.`Member ID`;**

When trying to update data from two tables an error message will occur as shown in the figure below:

Query:

**update view1 set `amount`= 300 , `Member ID`=54886660 where `amount`= 200 and
`Member ID`= 54886656;**

Error

SQL query:

```
update view1 set `amount`= 300 , `Member ID`=54886660 where `amount`= 200 and `Member ID`= 54886656
```

MySQL said: ?

```
#1393 - Can not modify more than one base table through a join view 'riverside club.view1'
```

RULE 7: HIGH-LEVEL INSERT, UPDATE, AND DELETE

Data can be retrieved from a relational database in sets constructed of data from multiple rows and/or multiple tables.

Evidence of satisfaction of rule 7:

To show that this rule is satisfied by the DBMS that I am using in this project, I carry out two tests. The first one consists of making changes on multiple rows in a single command and the second one consists of making changes in multiple tables in a single command.

a. Updating multiple rows by a single command:

We can update multiple rows of a table in a single command as follows:

Query:

update payments set `Payment Method`= 'payment by card' where `Payment Method`='cash';

➔ Changes are made in the table “payments” in multiple rows: this is shown on the following two figure, the first shows multiple rows with payment method = ‘cash’ and after this query, the second figure shows those lines with payment method changed to ‘payment by card’.

	Payment Reference	Member ID	Club ID	Payment Method	Amount	Payment Date	Annual Membership Renewal Date
  	P0020288	54886657	NULL	payment by card	200	2015-01-31	2016-01-31
  	P0020289	54886658	NULL	cash	200	2015-01-03	2016-01-03
  	P0020290	54886656	NULL	Cash	200	2015-02-03	2016-02-03
  	P0020291	NULL	C015	payment by card	500	2015-06-30	NULL

Options	Payment Reference	Member ID	Club ID	Payment Method	Amount	Payment Date	Annual Membership Renewal Date
	P0020288	54886657	NULL	payment by card	200	2015-01-31	2016-01-31
	P0020289	54886658	NULL	payment by card	200	2015-01-03	2016-01-03
	P0020290	54886656	NULL	payment by card	200	2015-02-03	2016-02-03
	P0020291	NULL	C015	payment by card	500	2015-06-30	NULL
	P0020292	54886661	NULL	200	NULL	NULL	NULL

b. Updating 2 tables using a single command:

The following query allows to update two tables in a single command. This query consists of changing the name of a club from the 'sporting associations' table and change its corresponding payment method in the table 'payments'.

Query:

update `sporting associations`

join `payments` on `sporting associations`.`Club ID`= payments.`Club ID`

set `sporting associations`.`Club Name`='Galway Club' , payments.Amount=600

where `sporting associations`.`Club ID`='C015';

Changes made in the table "payments": see the last row that contains 600 (was 500 before this query.)

Options	Payment Reference	Member ID	Club ID	Payment Method	Amount	Payment Date	Annual Membership Renewal Date
	P0020288	54886657	NULL	payment by card	200	2015-01-31	2016-01-31
	P0020289	54886658	NULL	payment by card	200	2015-01-03	2016-01-03
	P0020290	54886656	NULL	payment by card	200	2015-02-03	2016-02-03
	P0020291	NULL	C015	payment by card	600	2015-06-30	NULL

Changes made in the table "Sporting Associations": only one row is in this table, the name of the club was 'Galway Sports Club' and after the update query it became 'Galway Club'.

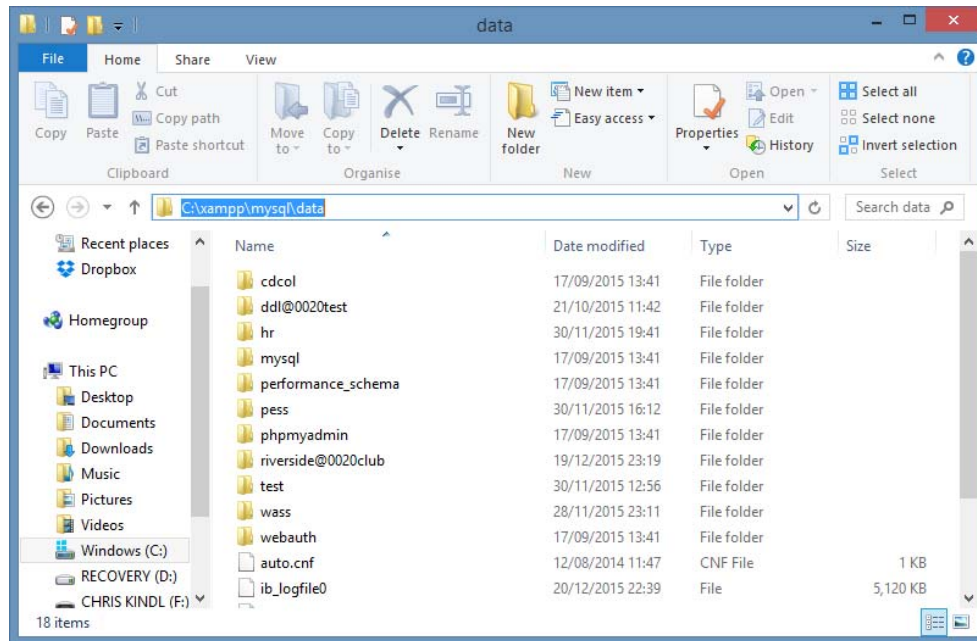
Options	Club ID	Club Name	Contact Details
	C015	Galway Club	871519785

RULE 8: PHYSICAL DATA INDEPENDENCE

“Applications programs remain unimpaired whenever any changes are made in either: Storage, representation or access method.”

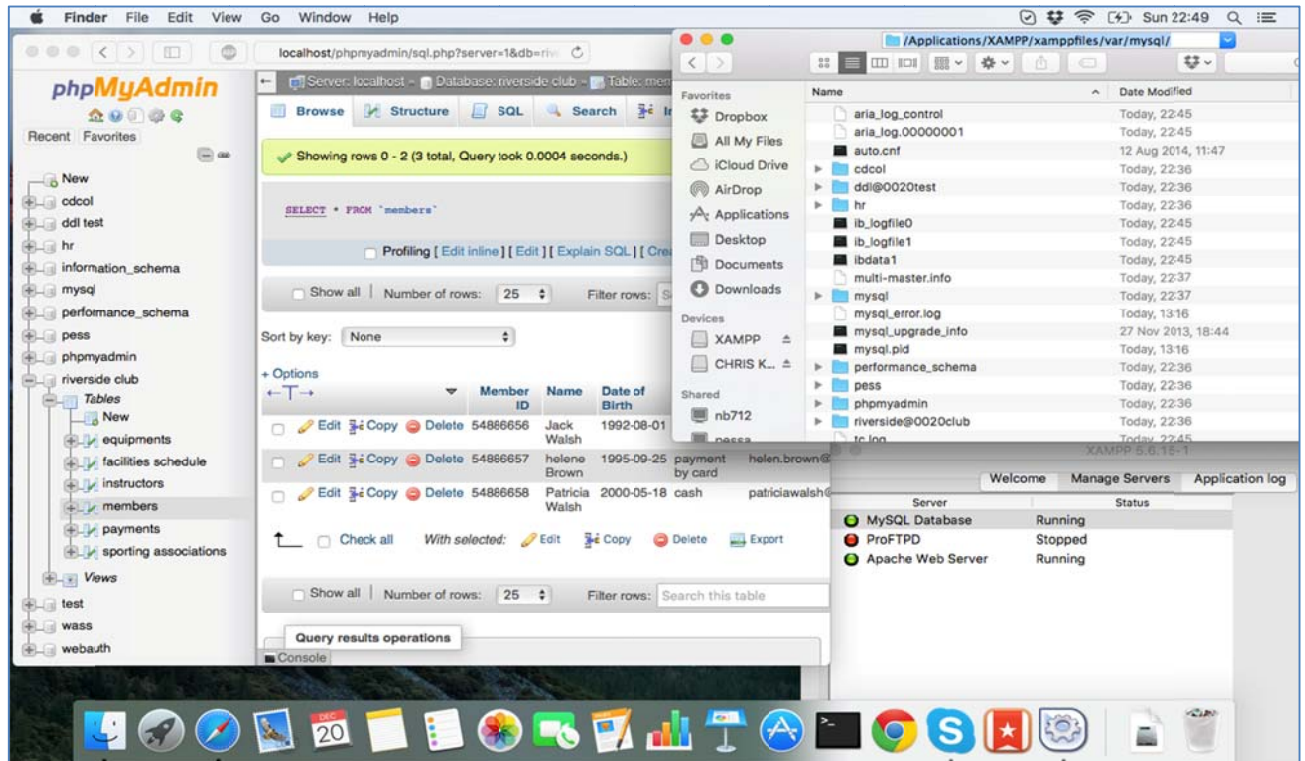
Evidence of satisfaction of rule 8 in storage, representation and access method:

In order to show that the physical storage, representation and access method to the database is independent, I copied the content of C:\xampp\mysql\data (see the following figure) into another machine Mac Os machine after installing XAMPP in it.



The destination machine is a MAC, its disk structure is different and the installed XAMPP also organizes its files in a different way. The destination folder of data in this new machine is in /Application/XAMPP/xamppfiles/var/mysql.

After starting the mysql server and opening the phpmyadmin page, I was able to retrieve the data that has been created on the first machine, see the following figure for a visual indication.



RULE 9: LOGICAL DATA INDEPENDENCE

This rule shows how a user views data, should not change when the logical structure (tables structure) of the database changes.

Evidence of satisfaction of rule 9:

In order to show that this rule is satisfied, we can for example add another table to our database and show that the all related applications are running without any issues. In this evaluation, I will add a new attribute to one table and visualize the content of that table to show that this is operation preserves the data.

Query:

Alter table `sporting associations`

ADD `Contact Person Name` VARCHAR(30) NULL DEFAULT NULL AFTER `Contact Details`;

Table "Sporting Associations" before adding an extra column:

	Club ID	Club Name	Contact Details
	C015	Galway Sports Club	871519785

Table "Sporting Associations" after adding a column 'Contact Person Name':

Options	Club ID	Club Name	Contact Details	Contact Person Name
<input type="checkbox"/> Edit <input type="checkbox"/> Copy <input type="checkbox"/> Delete	C015	Galway Sports Club	871519785	NULL

This shows that the data that was initially available in the table was kept even after changing the structure of the table by adding a new attribute.

RULE 10: INTEGRITY INDEPENDENCE

This rule is divided into two parts:

a. Entity integrity: no component of a primary key is allowed to have a null value

This feature is fully covered by MYSQL and this has already been discussed previously in Rule 3.

b. Referential integrity: for each distinct non-null foreign key value in a relational database, there must exist a matching primary key value from the same domain

The second part of the rule is partially fulfilled by the first part of the rule. Indeed, in any case a primary key cannot have a null value. With regard to foreign keys, MYSQL allows to have null values in foreign keys. This is shown in my database in table payments, where the two foreign keys, member id and club id, can be null. This is interpreted as follows (see the following figure): if a member id is null this means the payment is not related an individual member, and consequently there should be a value in the club id field and vice-versa.

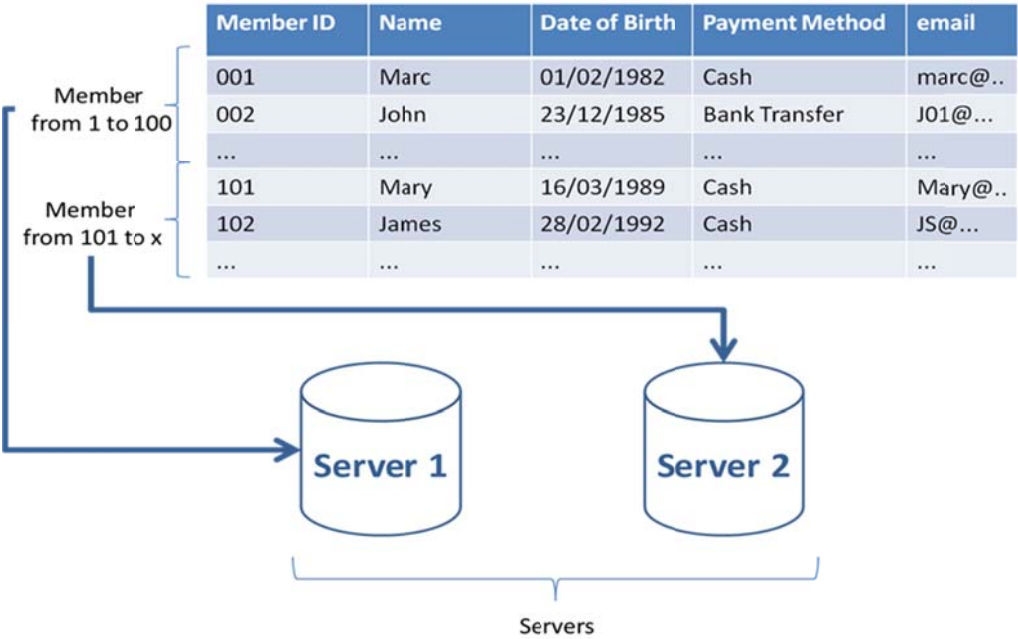
Options	Payment Reference	Member ID	Club ID	Payment Method	Amount	Payment Date	Annual Membership Renewal Date
<input type="checkbox"/> Edit <input type="checkbox"/> Copy <input type="checkbox"/> Delete	P0020288	54886657	NULL	payment by card	200	2015-01-31	2016-01-31
<input type="checkbox"/> Edit <input type="checkbox"/> Copy <input type="checkbox"/> Delete	P0020289	54886658	NULL	Cash	200	2015-01-03	2016-01-03
<input type="checkbox"/> Edit <input type="checkbox"/> Copy <input type="checkbox"/> Delete	P0020290	54886656	NULL	Cash	200	2015-02-03	2016-02-03
<input type="checkbox"/> Edit <input type="checkbox"/> Copy <input type="checkbox"/> Delete	P0020291	NULL	C015	payment by card	500	2015-06-30	NULL

RULE 11: DISTRIBUTION INDEPENDENCE

A database management system should be able to operate over multiple databases that are distributed over multiple servers. The distribution can be done vertically or horizontally.

Suggested horizontal distribution: with regard to my project, I can horizontally distribute my data over multiple servers as follows: divide the data horizontally using a split strategy that is suitable. For example, I suggest dividing each set of 100 members in one

server. As shown in the following figure, the first 100 members can be stores in a first server (server 1) and the following members on another server (server 2).



Suggested vertical distribution: with regard to my project, I can vertically distribute my data over multiple servers as follows: divide the data vertically using a split strategy that is suitable. For example, I suggest splitting the members table into two tables, the first one containing Member ID, Name and Date of Birth and the second one containing Member ID, Payment Method and email. As shown in the following figure, each table can be stored in a separate server.

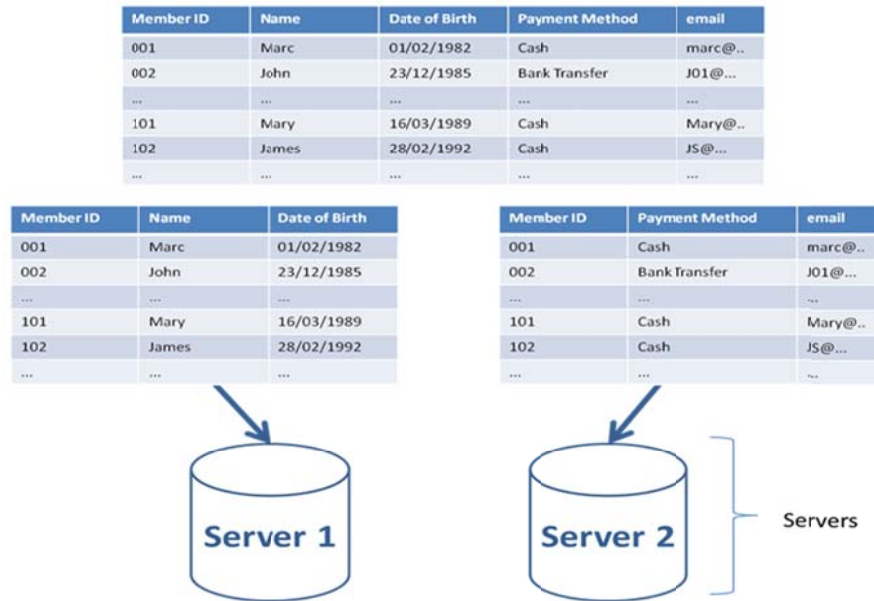
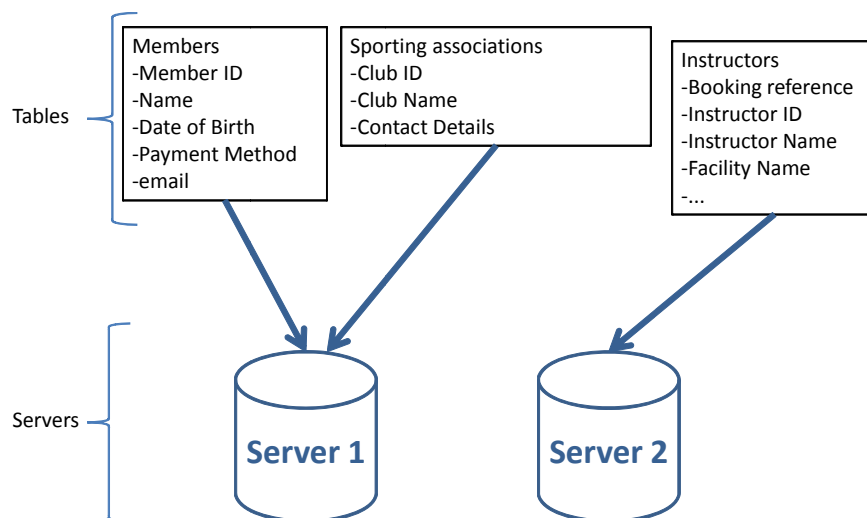


Table-based distribution: I also propose the possibility to store each table in a separate server; this can be seen as special case of the vertical distribution. As shown in the following figure, each set of tables can be stored on a specific server.



This rule proposes to distribute the data according to a strategy that satisfies the need of the database developers and users. MYSQL cluster is a tool that allows for such feature.