SQL MODULE

PREFACE

This module is prepared to be used as a one credit course. We assume the students have some background in relational database modeling. The main objective of this module is to teach fundamental knowledge of SQL (pronounced "es-sequel or sequel") stands for Structured Query Language.

LEARNING OBJECTIVES

1. Introducing the SQL system.
2. Learning how to create and maintain databases and tables with SQL statements.
3. Learning essential skills of SQL statements
4. Learning Intermediate skills of SQL.

SOFTWARE REQUIREMENTS

Every SQL database follows mostly standard SQL, but also has supplemental syntax, features, functions, and idiosyncrasies. There are several Relational Database Management System(RDBMS) implementations including:

1. MySQL
2. SQLite

For complete utilization of practical exercises and examples in the module you must use one of the above Database Management System. To introduce the concept of SQL, in chapter 1 we used MySQL RDBMS. In the following chapters we will be using SQLite RDBMS. To go along with the examples and exercises in the module, we strongly recommend the installation SQLite manager.

Unlike MySQL or SQL Server, the SQLite engine has no standalone processes with which the application program communicates. Instead, the SQLite library is linked in and thus becomes an integral part of the application program. The library can also be called dynamically. The application program uses SQLite's functionality through simple function calls, which reduce latency in database access: function calls within a single process are more efficient than inter-process communication. SQLite stores the entire database (definitions, tables, indices, and the data itself) as a single cross-platform file on a host machine.

Due to the server-less design, SQLite applications require less configuration than client-server databases. SQLite is called *zero-conf* because it does not require service management (such as startup scripts) or access control based on GRANT and passwords. Access control is handled by means of file system permissions given to the database file itself. SQLite is not the preferred choice for write-intensive deployments. However, for simple queries with little concurrency, SQLite performance profits from avoiding the overhead of passing its data to another process.

The installation and configuration guidelines of all the three RDBMS are outlined in appendix A.

TABLE OF CONTENTS

Chapter 1. Introduction to the SQL system

This chapter provides a short description of SQL, its origins, basic concepts and components, and some examples.

1.1. Overview of SQL Query Language

What is SQL?

SQL stands for Structured Query Language. It is the standard means of manipulating and querying data in relational databases. Basically, every relational DBMS (RDBMS) implements SQL, although with slight variations.

SQL can be divided into three subsets:

- **The Data Definition Language (DDL),** which includes commands for creating schema objects (tables, views etc) and manipulating them. It deals only with the metadata.
- **The Data Manipulation Language (DML),** which includes the commands for changing or retrieving the actual data.
- **The Data Control Language (DCL),** which includes the commands for database administration; that is, creating users, assigning permissions.

SQL commands are English based. SQL is NOT case sensitive, except for data inside quotes; however, there is a strong stylistic convention that requires writing keywords in all uppercase. Character constants are enclosed in SINGLE QUOTES, while double quotes are reserved for spelling names which contain special characters.

Although SQL statements do NOT need to end in a semicolon, most SQL interpreters use this character (;) to represent the end of a statement. That way you can use line breaks to make your statement more readable. SQL ignores extra white spaces and line breaks.

SQL Origin

Edgar F. Codd. Edgar Frank "Ted" Codd (19 August 1923 – 18 April 2003) was a computer scientist who used to work for IBM. He invented the relational model for database management, the theoretical basis for relational databases and relational database management systems. IBM commissioned a group to build a prototype based on Codd's ideas. This group created a simplified version of DSL/Alpha that they called SQUARE. Refinements, to SQUARE led to a language called SEQUEL, which was, finally, renamed SQL

1.2. The Data Definition Language (DDL)

Description:

A data definition language or data description language (DDL) is a syntax similar to a computer programming language for defining data structures, especially database schemas.
Many data description languages use a declarative syntax to define columns and data types. Structured query language (e.g., SQL), however, uses a collection of imperative verbs whose effect is to modify the

schema of the database by adding, changing, or deleting definitions of tables or other elements. These statements can be freely mixed with other SQL statements, making the DDL not a separate language.

Example:

```
CREATE TABLE courses (
  courseID      INT NOT NULL    AUTO_INCREMENT,
  courseName    CHAR(9)         NOT NULL,
  courseFee     FLOAT(20)       NOT NULL,
  maximumLimit  INTEGER(20)    NOT NULL,
  enrollment    BOOLEAN    NOT NULL,
  PRIMARY KEY (courseID),
  UNIQUE INDEX courseName (courseName)
);
```

Explanation:

The above code creates a table called courses. Five columns titled coursed, courseName, courseFee, maximumLimit, and enrollment are defined with INT and VARCHAR data types. Coursed is used as a primary key while courseName is used as an index of the courses table.

1.3. The Data Manipulation Language (DML)

Description:

SQL's DML includes statements to do alter the rows in a table, and to get data from one or more tables. Notice that these statements do NOT alter the schema at all, only the data in the table. DMLs may serve for purposes like:

- Inserting records into a table
- Deleting records from table
- Updating data
- Selecting some parts of the record

Example:

```
INSERT INTO courses (courseID, courseName, courseFee, maximumLimit, enrollment) VALUES
(1, 'ICS-140', 800.75, 5, True);
```

Explanation:

The above code inserts data into the first row of the courses table. Inserting these data never change the schema of the table. New entries will be automatically recorded into the next row.

1.4. The Data Control Language (DCL)

Description:

A data control language (DCL) is a syntax similar to a computer programming language used to control access to data stored in a database (Authorization). It is a component of SQL. Examples of DCL commands include: GRANT to allow specified users to perform specified tasks.

Example:

```
GRANT INSERT
ON *
TO admin@localhost
IDENTIFIED BY 'pass1234';
```

Explanation:

The above code provides the privilege to inserts records into all the database tables for the admin. N.B. the admin will be authenticated for password and the password entry must match “pass1234” to get access.

1.5. Data types in SQL

Common Datatypes

Although the full set of datatypes available varies with each DBMS, the following datatypes are among the most useful and commonly available:

- CHARACTER [(length)] or CHAR [(length)]
- VARCHAR (length)
- BOOLEAN
- SMALLINT
- INTEGER or INT
- DECIMAL [(p[,s])] or DEC [(p[,s])]
- NUMERIC [(p[,s])]
- REAL
- FLOAT(p)
- DOUBLE PRECISION
- DATE
- TIME
- TIMESTAMP
- CLOB [(length)] or CHARACTER LARGE OBJECT [(length)] or CHAR LARGE OBJECT [(length)]
- BLOB [(length)] or BINARY LARGE OBJECT [(length)]

1.6. Constraints

Constraints

In SQL we can attach constraints to each column or field, and we can also attach constraints to the whole table. The table constraints are added after all fields in the CREATE TABLE statement. For example, we can write the book table from Example 1, as follows:

Example: CREATE TABLE statement with constraints

```
CREATE TABLE courses(
courseId INT NOT NULL AUTO_INCREMENT,
courseName CHAR(9) NOT NULL,
courseFee FLOAT(20) NOT NULL,
maximumLimit INTEGER(20) NOT NULL,
enrollment BOOLEAN NOT NULL,
PRIMARY KEY (courseId),
UNIQUE INDEX courseName (courseName)
);
```

Notice that this notation allows us to give a name to each constraint (courseId and courseName in our case), and also that NOT NULL has to be expressed as a column constraint.

SQL constraints include the following:

- NOT NULL
- PRIMARY KEY
- UNIQUE
- FOREIGN KEY ... REFERENCES this constraint allows us to express a foreign key.
- CHECK This constraint allows us to add an arbitrary predicate to be tested

Chapter 2. SQL statements to create and manage database and database tables

N.B. First thing is first, before you proceed with this lesson make sure that you have installed SQLite manager. If you haven't done so, you may refer Appendix 1. A.

To check the correct installation of SQLite, from the command line, change your directory to the directory of SQLite installation. The SQLite project provides a simple command-line program named **sqlite3** (or **sqlite3.exe** on Windows) that allows the user to manually enter and execute SQL statements against an SQLite database. This document provides a brief introduction on how to use the **sqlite3** program.

2.1 How to start a database in SQLite:

To start the **sqlite3** go to the directory where you extracted the SQLite files.

```
C:\>cd sqlite

C:\sqlite>dir
 Volume in drive C is Windows
 Volume Serial Number is 1494-BC76

 Directory of C:\sqlite

08/01/2018  09:39 PM    <DIR>          .
08/01/2018  09:39 PM    <DIR>          ..
08/01/2018  09:31 PM                 0 database.db
08/01/2018  09:39 PM                 0 my_database;
06/04/2018  02:52 PM           461,824 sqldiff.exe
06/04/2018  02:53 PM           871,936 sqlite3.exe
06/04/2018  02:53 PM         1,974,784 sqlite3_analyzer.exe
               5 File(s)      3,308,544 bytes
               2 Dir(s)  892,371,574,784 bytes free

C:\sqlite>
```

Make sure that ***sqlite3.exe*** exists

2.2 SQL code to create a database

- **using SQLite Syntax**

To create a new SQLite database named "my database", type the following code on command line

```
C:\sqlite>sqlite3 my_database.db
```

Example:

```
C:\>sqlite3 my_database.db
SQLite version 3.23.1 2018-04-10 17:39:29
Enter ".help" for usage hints.
sqlite>
```

Note. The above statement will create a database with the name **"my_database".** If the database name does not exist, a new database file with the given name will be created automatically. The command line changes to sqlite > prompt. SQLite doesn't use the CREATE DATABASE statement like other databases.

To show the created database type the following code at the command line

```
sqlite>.show
```

Example:

```
sqlite> .show
        echo: off
         eqp: off
     explain: auto
     headers: off
        mode: list
   nullvalue: ""
      output: stdout
colseparator: "|"
rowseparator: "\n"
       stats: off
       width:
    filename: my_database.db
sqlite>
```

- **using MySQL Syntax**

  MySQL database uses the CREATE DATABASE statement like other databases.

  To create a new MySQL database named "my database", type the following code on command line

  ```
  mysql> CREATE DATABASE courses_db;
  ```

Example:

```
MariaDB [(none)]> CREATE DATABASE courses_db;
Query OK, 1 row affected (0.02 sec)
```

SQLite Meta Commands:

Meta Commands are used for definition and administration operations. In SQLite, they always start with a dot. Here are some of the common ones:

Command	Description
.show	Displays current settings for various parameters
.databases	Provides database names and files
.quit	Quit sqlite3 program
.tables	Show current tables
.schema	Display schema of table
.header	Display or hide the output table header
.mode	Select mode for the output table
. dump	Dump database in SQL text format

2.3 How to create table

To create a table in a database named, type the following code on command line

- **SQLite syntax**

```
sqlite>CREATE TABLE “table_name” (table columns);

Example:
CREATE TABLE courses(
  courseId INT PRIMARY KEY NOT NULL,
  courseName          CHAR(9) NOT NULL,
  courseFeeREAL NOT NULL,
  maximumLimit INT    NOT NULL,
  enrollment  BOOLEAN
);
```

- **MySQL syntax**

```
CREATE TABLE courses (
courseId INT NOT NULL AUTO_INCREMENT,
courseName CHAR(9) NOT NULL,
courseFee FLOAT(20) NOT NULL,
maximumLimit INTEGER(20) NOT NULL,
enrollment BOOLEAN NOT NULL,
PRIMARY KEY (courseId),
UNIQUE INDEX courseName (courseName)
);
```

Note. The above SQL code will create a “courses” table that has five columns named “courseId, courseName, courseFee, maximumLimit, and enrollment. “

Example:

```
sqlite> CREATE TABLE courses(
   ...>    courseId INT PRIMARY KEY NOT NULL,
   ...>    courseName    CHAR(9) NOT NULL,
   ...>    courseFee     REAL NOT NULL,
   ...>    maximumLimit INT     NOT NULL,
   ...>    enrollment    BOOLEAN
   ...> );
sqlite> .tables
courses
sqlite>
```

Note. To show the current table use the. tables SQL command.

2.4 Drop table statement

Note: To remove a table from the database you use SQL DROP statement.

- **SQLite/MySQL DROP TABLE statement.**

```
DROP TABLE table_name
```

Example:

```
sqlite> DROP TABLE courses;
sqlite> .tables
sqlite>
```

Note. After we run the DROP TABLE SQL statement, running the. tables commands view no table. This confirms the removal of courses table from the database.

```
TRUNCATE TABLE table_name;
```

Note: The TRUNCATE TABLE statement is used to delete the data inside a table, but not the table itself.

2.5 Drop database statement

Note. SQLite does not use the DROP DATABASE statement like many other database management systems do. To drop the database in SQLite. You just have to delete the file manually.

- **MySQL DROP database command**

```
DROP DATABASE database_name;
```

Example:

```
MariaDB [courses_db]> drop database courses_db;
Query OK, 1 row affected (0.27 sec)

MariaDB [(none)]>
```

Chapter 3. Essential skills of SQL statement

3.1 Describe statement

The DESCRIBE statement displays metadata about a table, such as the column names and their data types.

- **SQLite syntax**

  SQLite does not use the DESCRIBE table statement like other database management systems do. But, we can use ".schema" SQLite command to display the metadata about a table.

```
.schema 'tablename'
```

Example:

```
sqlite> .schema courses
CREATE TABLE courses(
   courseId INT PRIMARY KEY NOT NULL,
   courseName   CHAR(9) NOT NULL,
   courseFee    REAL NOT NULL,
   maximumLimit INT     NOT NULL,
   enrollment   BOOLEAN
);
sqlite>
```

- **MySQL syntax**

```
DESCRIBE table_name;
```

Example:

```
MariaDB [courses_db]> describe courses;
+--------------+------------+------+-----+---------+----------------+
| Field        | Type       | Null | Key | Default | Extra          |
+--------------+------------+------+-----+---------+----------------+
| courseId     | int(11)    | NO   | PRI | NULL    | auto_increment |
| courseName   | char(9)    | NO   | UNI | NULL    |                |
| courseFee    | float      | NO   |     | NULL    |                |
| maximumLimit | int(20)    | NO   |     | NULL    |                |
| enrollment   | tinyint(1) | NO   |     | NULL    |                |
+--------------+------------+------+-----+---------+----------------+
5 rows in set (0.05 sec)

MariaDB [courses_db]>
```

Note. The courses table displays Field(columns) of the table, Type, Key, and other parameters based on the setting of the table.

3.2 Select statement

The SELECT statement is used to select data from a database. If you want to select all the fields available in the table, instead of column name use an asterisk (*)

- **SQLite/MySQL SELECT statement.**

```
SELECT column1, column2, ...
FROM table_name;
```

Example: SQLite:

```
sqlite> select * from courses;
1|ICS-14|800.56|5|1
2|ICS-24|999.99|4|1
3|BIO-100|777.99|5|0
sqlite>
```

Example: MySQL database:

```
sqlite> select * from courses;
1|ICS-14|800.56|5|1
2|ICS-24|999.99|4|1
3|BIO-100|777.99|5|0
sqlite>
```

3.3 WHERE clause

The SQL WHERE clause is used to filter the results and apply conditions in a SELECT, INSERT, UPDATE, or DELETE statement.

- **The syntax for the WHERE clause in SQL is:**

```
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

conditions

The conditions that must be met for records to be selected.

Example:

```
sqlite> SELECT courseName from courses where courseId = 2;
ICS-24
sqlite>
```

Note: The course name (ICS-24) with the courseId of 2 is selected.

3.4 SQL Insert, update, delete commands

The INSERT INTO statement of SQL is used to insert a new row in a table.

- **The general syntax of INSERT statement is:**

```
INSERT INTO table_name (column1, column2, column3,..)
VALUES (value1, value2, value3,..);
WHERE condition;
```

Example:

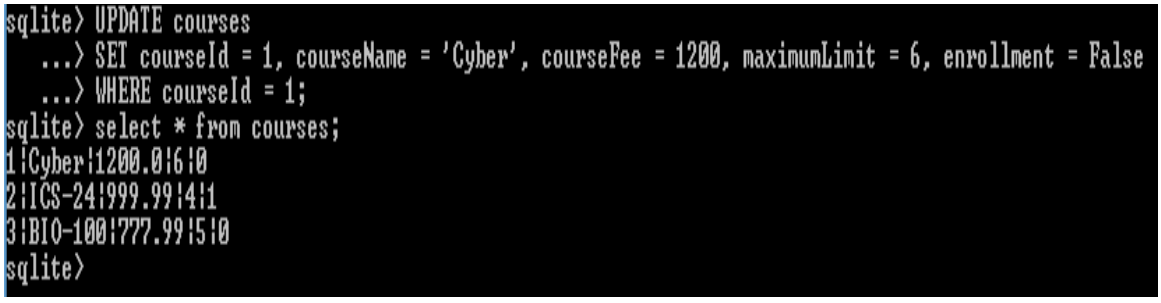
```
sqlite> INSERT INTO courses(courseID, courseName, courseFee, maximumLimit, enrollment)
   ...> VALUES (1, 'ICS-14', 800.56, 5, True);
sqlite> select * from courses;
1|ICS-14|800.56|5|1
```

The UPDATE statement is used to modify the existing records in a table.

- **The general SQL syntax for the UPDATE statement is:**

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

Example:



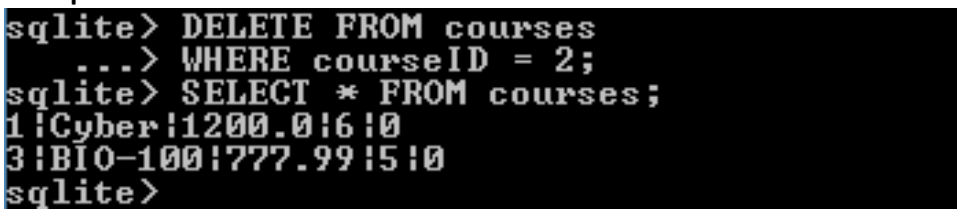
Note: The table is update with a new courseName and a new enrollment records.

The DELETE FROM statement in SQL is used to remove records from a table. Make sure that a condition is specified, otherwise all records will be removed.

- **The general syntax of DELETE statement is:**

```
DELETE FROM table_name
WHERE condition;
```

Example:



Note: Be careful when deleting records in a table! You will delete any information in the table as well. You won't normally be asked to confirm.

3.5 Order by clause

The **ORDER BY** keyword is used to sort the result-set in ascending or descending order

The **ORDER BY** keyword sorts the records in ascending order by default. To sort the records in descending order, use the DESC keyword.

- **The general ORDER BY Syntax:**

```
SELECT column1, column2, ...
FROM table_name
ORDER BY column1, column2, ... ASC|DESC;
```

Example:

3.6 Aggregate

SQL aggregate functions return a single value, calculated from values in a column

- **SQL aggregate to find average value:**

```
SELECT AVG(column_name)
FROM table_name
WHERE condition;
```

Example:

3.7 Group by

The GROUP BY statement is often used with aggregate functions (COUNT, MAX, MIN, SUM, AVG) to group the result-set by one or more columns.

- **SQL Group by statement**

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
ORDER BY column_name(s);
```

Example:

```
sqlite> SELECT courseName
   ...> FROM courses
   ...> WHERE courseID = 2
   ...> GROUP BY courseName
   ...> ORDER BY courseName;
Chem-100
sqlite>
```

Chapter 4. Intermediate skills of SQL statement

4.1. Join Statements

The SQL Joins statement is used to combine records from two or more tables in a database. A JOIN is a means for combining fields from two tables by using values common to each.

Consider the following two tables

Course Table

Id	Course ID	Course Name	Course Fee	Maximum Limit	Enrollment
1	Chem-100	Intro to Chemistry	555.99	5	False
2	ICS-140	Programming 1	888.99	30	True
3	ICS-141	Programming 2	999.99	30	True
4	ICS-240	Programming 3	1200.00	30	True

Student table

Id	Student ID	First Name	Last Name
1	1001	Sara	Johnson
2	1002	John	Mike
3	1003	Alex	Adam
4	1004	Moha	Abdi

We can join the two tables different ways. For example, in our example database, there are two tables: Courses and Student table. The contact table contains the column Coursed, which correlates with the Primary-Key column id of the Course table. By evaluating the column values, we can join courses and Students table together.

Run the following SQL statement and as result you will get the joined table.

```
SELECT *
FROM   courses c
JOIN   students s ON c.courseId = s.courseId;
```

Joined (virtual) table, created out of Courses & Student

Id	Course ID	Course Name	Course Fee	Maximum Limit	Enrollment	Student ID	First Name	Lat Name	Course ID
1	Chem-100	Intro to Chemistry	555.99	5	False	1001	Sara	Johnson	Chem-100
2	ICS-140	Programming 1	888.99	30	True	1002	John	Mike	ICS-140
3	ICS-141	Programming 2	999.99	30	True	1003	Alex	Adam	ICS-141
4	ICS-240	Programming 3	1200.00	30	True	1004	Moha	Abdi	ICS-240

Different types of SQL joins

SQLite doesn't support different types of SQL joins, so in this section we are using MySQL DBMS to demonstrate the properties of different types of SQL joins.

Types of SQL joins include:

- INNER JOIN
- LEFT JOIN
- RIGHT JOIN
- FULL OUTER JOIN(UNION)

Use the source code included in this module to create the following **courses** and **students** tables in MySQL database.

Courses table

starId	courseId	courseName	courseFee	maximumLimit	enrollment
1	CHE-100	Intro to Chemstry	555.55	25	1
2	ICS-140	Programming 1	888.99	30	1
3	ICS-141	Programming 2	999.99	25	1
4	ICS-240	Programming 3	1200	20	1
5	Bio-100	Intro to Biology	544.99	5	1
6	CFS-100	Intro to FS	1300	25	1

Students table

studentID	firstName	lastName	starId
1001	Sara	Johnson	1
1002	John	Mike	2
1003	Alex	Adam	3
1004	Moha	Abdi	4
1005	Martha	Alex	5

INNER JOIN:

General SQL Syntax for INNER JOIN.

```
SELECT column_name(s)
FROM table1
INNER JOIN table2 ON table1.column_name = table2.column_name;
```

Example

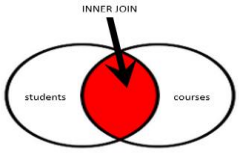
```
--inner join--
SELECT *
FROM courses
INNER JOIN students ON courses.starId = students.starId;
```

Query Output from inner join

starId	courseId	courseName	courseFee	maximumLimit	enrollment	studentID	firstName	lastName	starId
1	CHE-100	Intro to Chemstry	555.55	25	1	1001	Sara	Johnson	1
2	ICS-140	Programming 1	888.99	30	1	1002	John	Mike	2
3	ICS-141	Programming 2	999.99	25	1	1003	Alex	Adam	3
4	ICS-240	Programming 3	1200	20	1	1004	Moha	Abdi	4
5	Bio-100	Intro to Biology	544.99	5	1	1005	Martha	Alex	5

Discussion

The INNER JOIN statement selects records that have matching values in both tables.



LEFT JOIN:

General SQL Syntax for LEFT JOIN.

```
SELECT column_name(s)
FROM table1
LEFT JOIN table2 ON table1.column_name = table2.column_name;
```

Example—LEFT JOIN--

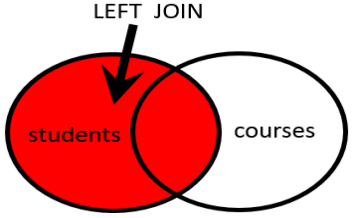
```
--LEFT JOIN--
SELECT *
FROM courses
LEFT JOIN students ON courses.starId = students.starId;
```

Query Output from LEFT JOIN

starId	courseId	courseName	courseFee	maximumLimit	enrollment	studentID	firstName	lastName	starId
1	CHE-100	Intro to Chemstry	555.55	25	1	1001	Sara	Johnson	1
2	ICS-140	Programming 1	888.99	30	1	1002	John	Mike	2
3	ICS-141	Programming 2	999.99	25	1	1003	Alex	Adam	3
4	ICS-240	Programming 3	1200	20	1	1004	Moha	Abdi	4
5	Bio-100	Intro to Biology	544.99	5	1	1005	Martha	Alex	5
6	CFS-100	Intro to FS	1300	25	1	NULL	NULL	NULL	NULL

Discussion

The LEFT JOIN query returns all records from the left table and the matched records from the right table. The result is NULL from the right side if there is no match.



RIGHT JOIN:
General SQL Syntax for RIGHT JOIN.

```
SELECT column_name(s)
FROM table1
RIGHT JOIN table2 ON table1.column_name = table2.column_name;
```

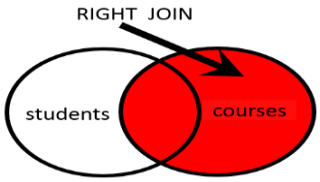
Example—LEFT JOIN--

```
--LEFT JOIN--
SELECT *
FROM courses
RIGHT JOIN students ON courses.starId = students.starId;
```

Query Output from RIGHT JOIN

starId	courseId	courseName	courseFee	maximumLimit	enrollment	studentID	firstName	lastName	starId
1	CHE-100	Intro to Chemstry	555.55	25	1	1001	Sara	Johnson	1
2	ICS-140	Programming 1	888.99	30	1	1002	John	Mike	2
3	ICS-141	Programming 2	999.99	25	1	1003	Alex	Adam	3
4	ICS-240	Programming 3	1200	20	1	1004	Moha	Abdi	4
5	Bio-100	Intro to Biology	544.99	5	1	1005	Martha	Alex	5

Discussion
The RIGHT JOIN keyword returns all records from the right table, and the matched records from the left table the result is NULL from the left side, when there is no match.



FULL OUTER JOIN(UNION) :
General SQL Syntax for RIGHT JOIN.

```
SELECT column_name(s) from table2
LEFT JOIN table1 ON table2.column_name = table1.column_name
UNION
SELECT column_name(s) table1
LEFT JOIN table2 ON table1.column_name = table2.column_name;
```

Example—LEFT JOIN--

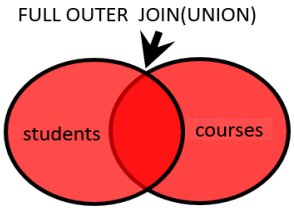
```
--FULL OUTER JOIN (UNION)--
SELECT * FROM students
LEFT JOIN courses ON students.starId = courses.starId
UNION
SELECT * FROM courses
RIGHT JOIN students ON courses.starId = students.starId;
```

Query Output from FULL OUTER JOIN(UNION)

studentID	firstName	lastName	starId	starId	courseId	courseName	courseFee	maximumLimit	enrollment
1001	Sara	Johnson	1	1	CHE-100	Intro to Chemstry	555.55	25	1
1002	John	Mike	2	2	ICS-140	Programming 1	888.99	30	1
1003	Alex	Adam	3	3	ICS-141	Programming 2	999.99	25	1
1004	Moha	Abdi	4	4	ICS-240	Programming 3	1200	20	1
1005	Martha	Alex	5	5	Bio-100	Intro to Biology	544.99	5	1
1	CHE-100	Intro to Chemstry	555.55	25	1	1001	Sara	Johnson	1
2	ICS-140	Programming 1	888.99	30	1	1002	John	Mike	2
3	ICS-141	Programming 2	999.99	25	1	1003	Alex	Adam	3
4	ICS-240	Programming 3	1200	20	1	1004	Moha	Abdi	4
5	Bio-100	Intro to Biology	544.99	5	1	1005	Martha	Alex	5

Discussion

The FULL OUTER JOIN keyword return all records when there is a match in either left or right table records.



4.2. Views

Views in SQL are virtual tables created from database tables. In a database, a view is the result set of a stored query on the data, which the database users can query just as they would in a persistent database collection object.

A view also has rows and columns as they are in a real table in the database. We can create a view by selecting fields from one or more tables present in the database.

Views can provide advantages over tables:

- Views can represent a subset of the data contained in a table. Consequently, a view can limit the degree of exposure of the underlying tables to the outer world: a given user may have permission to query the view, while denied access to the rest of the base table.
- Views can join and simplify multiple tables into a single virtual table.

- Views can act as aggregated tables, where the database engine aggregates data (sum, average, etc.) and presents the calculated results as part of the data.
- Views can hide the complexity of data.
- Views take very little space to store; the database contains only the definition of a view, not a copy of all the data that it presents.
- Depending on the SQL engine used, views can provide extra security.

General SQL Syntax for RIGHT JOIN.

```
CREATE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

Example

```
CREATE VIEW ics240_view AS
SELECT *
FROM courses
WHERE courses. courseID = 'ICS-240';
SELECT * FROM ics240_view;
```

Output from the view query

```
MariaDB [courses_db]> SELECT * FROM ics240_view;
+--------+----------+---------------+-----------+--------------+------------+
| starId | courseId | courseName    | courseFee | maximumLimit | enrollment |
+--------+----------+---------------+-----------+--------------+------------+
|      4 | ICS-240  | Programming 3 |      1200 |           20 |          1 |
+--------+----------+---------------+-----------+--------------+------------+
```

Discussion

As you see from the output queries, the view for 'isc240_view' is created. Like a regular table, we used the SELECT * query to show the record in the view.

4.3. SQL Integrity constraints

In SQL, Integrity Constraints are used to specify business rules for the database tables.
Constraints are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the table. If there is any violation between the constraint and the data action, the action is aborted. Constraints can be column level or table level. Column level constraints apply to a column, and table level constraints apply to the whole table.

Column Constraint

The column constraint clause specifies conditions which all values must meet. There are different column constraint types Including:

- NOT NULL
- Primary Key
- Unique
- Foreign Key
- Check values

The **NOT NULL** phrase defines, that it is not allowed to store the **NULL** value in the column.

In the above examples, starId INT PRIMARY KEY NOT NULL AUTO_INCREMENT ,

- The starId is per definition not allowed to hold the Null value.

```
-- This INSERT command will fail
INSERT INTO t1(col_1) values(NULL);

-- The same applies to the following UPDATE command
INSERT INTO t1(col_1) values(5);
UPDATE t1 SET col_1 = NULL;
```

The **PRIMARY KEY** phrase defines that the column acts as the Primary Key of the table. This implies that the column is not allowed to store a **NULL value** and that the values of all rows are distinct from each other.

```
CREATE TABLE courses(
  starId INT PRIMARY KEY NOT NULL AUTO_INCREMENT ,

-- This INSERT will fail because a primary key column is not allowed to store the NULL value.
INSERT INTO t2(col_1) VALUES(NULL);

-- This INSERT works
INSERT INTO courses(starId) VALUES(5);

-- But the next INSERT will fail, because only one row with the value '5' is allowed.
INSERT INTO courses(starId) VALUES(5);
```

The **UNIQUE** constraint has a similar meaning as the PRIMARY KEY phrase. But there are two slight differences.

- First, the values of different rows of a UNIQUE column are not allowed to be equal, which is the same as with PK. But they are allowed to hold the NULL value, which is different from PK. The existence of NULL values has an implication. As the term *null = null* never evaluates to *true* (it evaluates to *unknown*) there may exist multiple rows with the NULL value in a column which is defined to be UNIQUE.
- Second, only one PK definition per table is allowed. In contrast, there may be many UNIQUE constraints (on different columns).

```
CREATE TABLE students (studentId VARCHAR UNIQUE);
```

```
-- works well
INSERT INTO students(studentId) VALUES('10as12');
-- repeating the same insert instruction is not allowed with the same student of '10as12'
INSERT INTO students(studentId) VALUES('10as12');
-- works well
INSERT INTO students(studentId) VALUES (null);
-- works also
INSERT INTO students(studentId) VALUES(null);
```

The **FOREIGN KEY** condition defines that the column can hold only those values, which are also stored in a different column of (the same or) another table. This different column has to be UNIQUE or a Primary Key, whereas the values of the foreign key column itself may hold identical values for multiple rows. The consequence is that one cannot create a row with a certain value in this column before there is a row with exactly this certain value in the referred table.

In our example database we have a *courses* table whose column *starID* as the primary key of the table. The starId is used as a foreign key in the student table.

```
CREATE TABLE students (
FOREIGN KEY (starId) REFERENCES courses(starId)
```

Column checks inspect the values of the column to see whether they meet the defined criterion. Within such column checks only the actual column is visible. If a condition covers two or more columns (e.g.: col_1 > col_2) a table check must be used.

Table Constraint

Table constraints defines rules which are mandatory for the table as a whole. Their sematic and syntax overlaps partially with the previous shown column constraints.

Table constraints are defined after the definition of all columns. The syntax starts with the key word CONSTRAINT and includes the possibility to denominate them with a meaningful name, *t6_pk*, *t6_uk* and *t6_fk* in the next example.

In the case of any exception most DBMS shows this name as part of the error message - and if you haven't defined one it uses its internal naming conventions which may be very cryptic.

Primary Key, UNIQUE and Foreign Key

In the same manner as shown in the column constraints part Primary Key, UNIQUE and Foreign Key conditions can be expressed as table constraints. The syntax differs slightly from the column constraint syntax, the semantic is identical.

As you have seen some constraints may be defined as part of the column definition, which is called a *column constraint*, or as a separate *table constraint*. Table constraints have two advantages.

First, they are a little bit more powerful.
Second, they do have their own name! This helps to understand system messages. Furthermore, it opens the possibility to manage constraints after the table exists and contains data. The ALTER TABLE statement can deactivate, activate or delete constraints. To do so, you have to know their name.

4.4. Transaction control

A transaction is an embracing of **one or more** SQL statements - especially of such statements, which write to the database such as INSERT, UPDATE or DELETE, but also the SELECT command can be part of a transaction. All writing statements must be part of a transaction. The purpose of transactions is the guarantee that the database changes only from one consistent state to another consistent state fading out all intermediate situations. This holds true also in critical situations such as parallel processing, disc crash, power failure, ... Transactions ensure the **database integrity**.

To do so they support four basic properties, which all in all are called the ACID paradigm.

Atomic	All SQL statements of the transaction take place or none.
Consistent	The sum of all data changes of a transaction transforms the database from one consistent state to another consistent state.
Isolated	The isolation level defines, which parts of uncommitted transactions are visible to other sessions.
Durable	The database retains committed changes even if the system crashes afterwards.

4.5. Normalization

Normalization provides a set of rules and patterns that can be applied to any database to avoid common logical inconsistencies. Normalizing a database design will typically improve

- Consistency, since errors that could be made with the database would be structurally impossible
- Extensibility, since changes to the database structure will only affect parts of the database they are logically dependent on
- Efficiency, since redundant information will not need to be stored

The database community has identified several distinct levels of normalization, each more stringent than the last. These are referred to as normal forms and are numbered from one (the lowest form of normalization, referred to as first normal form or 1NF) through five (fifth normal form or 5NF). It's quite common in practice to speak of one database design as being more or less normalized than another, as defined by these levels.

In practical applications, you'll often see 1NF, 2NF, and 3NF along with the occasional 4NF. Fifth normal form is very rarely seen.

Appendix.

A. How to install SQLite

How to Install SQLite On Windows: **https://youtu.be/wXEZZ2JT3-k**
How to Install SQLite On Mac: **https://youtu.be/iyXYwNQC6ag**
How to Install SQLite On Linux: **https://youtu.be/z94raoO7AII**

B. How to install MySQL

How to Install MySQL On Windows: **https://youtu.be/i-GlTXE2JU4**
How to Install MySQL On Mac: **https://youtu.be/iOlJxOkp6sI**
How to Install MySQL On Linux: **https://youtu.be/0o0tSaVQfV4**