

SQL 1: CREATE and DROP Tables

COMP1048: Databases and Interfaces (2024-2025)

Matthew Pike and Yuan Yao

Overview

In this lecture, we will cover:

- Review - What is a DBMS?
- Introduction to SQL - What can it do?
- Using SQL to:
 - **CREATE** tables in a database
 - Link tables with **FOREIGN KEY** constraints
 - **DROP** (delete) a table from a database

DBMSs and SQL

Database Management Systems (DBMS)

A DBMS is a collection of programs that allows users to create, manage, and interact with a database.

- A structured way to organise, store, and retrieve data
- A query language (often SQL) to interact with the database
- An administrative interface (often CLI) to manage the system
- A programmatic interface (API) for applications to access the database
- Essential functions like security, concurrency control, transaction management, and crash recovery to maintain data integrity
- Examples of DBMSs include:
 - SQLite
 - MariaDB
 - MySQL

SQL - Structured Query Language

- SQL is the standard language for managing data in a relational database, based on Edgar F. Codd's relational model (Codd, 1970).
- SQL is declarative, specifying *what* to do, not *how* to do it.
- Statements may not be executed in the order written, though there are rules about their declaration.

Example:

- `SELECT * FROM Student;`
 - Retrieves all data from the `Student` table.
- `SELECT * FROM Student WHERE SID < 100;`
 - Retrieves data from `Student` where `SID` is less than 100.

- Although SQL is a standard (ANSI in 1986, ISO in 1987), not all DBMSs implement it in the same way. SQL queries may need adjustments for different systems.
- For example:

“PostgreSQL folds unquoted names to lower case, contrary to the SQL standard, which specifies folding to upper case. Thus, Foo should match FOO not foo according to the standard.” (Wikipedia, 2023)

SQL consists of various operations for creating, selecting, updating, and removing data in a database. It can be informally divided into three sublanguages:

1. **Data Definition Language (DDL)** - used for creating and modifying database objects like tables and indices. DDL defines the structure and organisation of data.
2. **Data Manipulation Language (DML)** - used for inserting, retrieving, and modifying data.
3. **Data Control Language (DCL)** - used for managing security and concurrent access, including granting/revoking privileges and handling transactions.

Creating Tables with CREATE in SQL

Terminology

- We have discussed relational and ER representations of data.
- Now, we will explore how to implement these designs in a relational database using SQL.
- Table 1 provides a terminology mapping between different representations.

Relations	ER Diagrams (ERD)	Relational Databases
Relation	Entity	Table
Tuple	Instance	Row
Attribute	Attribute	Column/Field
Foreign Key	M:1 Relationship	Foreign Key
Primary Key	<u>Attribute</u>	Primary Key

Table 1: Terminology mapping between Relational, ERD, and Relational Databases

Going From ERD to Relational Databases using SQL

- **Goal:**
 - Given an ERD (e.g. Figure 1), create a relational database using SQL to represent the data structure.
- **Steps:**
 1. Translate entities into tables.
 2. Translate attributes into columns.
 3. Assign data types to columns to approximate attribute domains.
 4. Translate relationships into foreign keys.

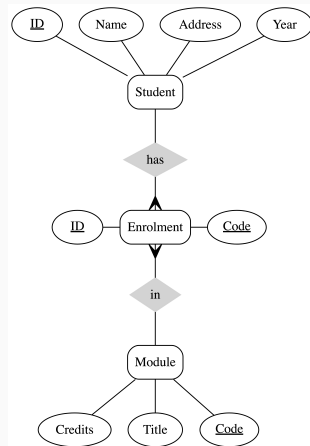


Figure 1: ERD for Student Module Enrolment

Example: Student Table

Aim: Create a table in SQL to represent the **Student** entity in Figure 2

- Student IDs must be unique and cannot be **NULL**.
- Addresses are optional and can be **NULL**.
- If not specified, the **Year** of study defaults to 1.

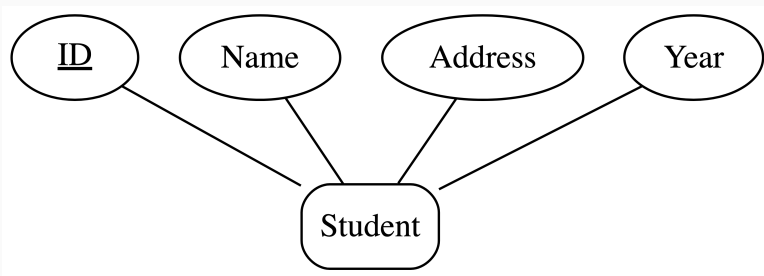



Figure 2: ER Diagram for Student Table

Step 1: Translate Entities to Tables

```
CREATE TABLE Student(  
    ...  
);
```

 Remember to use semicolons!

A semicolon ; is used at the end of each SQL statement to indicate the end of the statement.

Step 2: Attributes of an Entity become Columns

```
CREATE TABLE Student (  
    sID ,  
    sName,  
    sAddress,  
    sYear  
);
```



Caution with Commas

In SQL, you must separate columns with commas. However, you should not put a comma after the last column in the list.

Step 3: Assign Types to Columns

```
CREATE TABLE Student (  
    sID INTEGER,  
    sName VARCHAR(50),      -- Reasonable?  
    sAddress VARCHAR(255),  -- Reasonable?  
    sYear INTEGER  
);
```

Comments in SQL

Just as with other programming languages, SQL supports comments. Comments are ignored by the DBMS and are used to document your code.

- Single line comments start with --
- Multi-line comments start with /* and end with */

Step 4: Add constraints

```
CREATE TABLE Student (  
    sID INTEGER PRIMARY KEY,  
    sName VARCHAR(50) NOT NULL,  
    sAddress VARCHAR(255),  
    sYear INTEGER DEFAULT 1  
);
```

```
CREATE TABLE Student (  
    sID INTEGER,  
    sName VARCHAR(50) NOT NULL,  
    sAddress VARCHAR(255),  
    sYear INTEGER DEFAULT 1,  
    CONSTRAINT pk_student PRIMARY KEY (sID)  
);
```

Inline vs Named Constraints

In SQL, constraints can be defined inline with columns or as named constraints. Both methods are functionally identical, but named constraints (e.g. `pk_student`) can be referenced later, which is useful for managing multiple constraints.

- Constraints are a key part of database design, as they impose rules on the data stored in a table. These rules help ensure that the data remains consistent, accurate, and reliable.
- For instance, constraints can enforce that a column cannot have **NULL** values or that all entries must be **UNIQUE**.
- There are numerous types of constraints in SQL, including:
 - **PRIMARY KEY**
 - **FOREIGN KEY**
 - **UNIQUE**
 - **NOT NULL**

- A **PRIMARY KEY** uniquely identifies each row and cannot contain **NULL** values.
- A **UNIQUE** constraint ensures all values in a column are distinct but can contain **NULL**.



A Quirk with SQLite

SQLite allows **NULL** in **PRIMARY KEY** columns unless it's an **INTEGER PRIMARY KEY, WITHOUT ROWID** table, **STRICT** table, or the column is explicitly **NOT NULL**. However, for simplicity, it's best to think of **PRIMARY KEY** as generally not allowing **NULL** values.

[More info](#)

Types in SQL

Data Types are DBMS Dependent

Not all data types are supported by all DBMSs, and some may be implemented differently across systems.

- SQL offers a variety of data types for representing data in a database, including:
 - Numeric types: `INTEGER`, `REAL`, `NUMERIC`
 - Character types: `CHAR`, `VARCHAR(M)`, `NCHAR`, `NVARCHAR(M)`
 - String types: `VARCHAR`, `TEXT`
 - Date and time types: `DATE`, `TIME`, `TIMESTAMP`

Examples of Data Types in SQL

Data Type	Description	Example
INTEGER	Whole numbers	1, 2, 3
REAL	Floating point (decimal) numbers	1.0, 2.5, 3.1415
CHAR(n)	Fixed-length string of n characters	CHAR(2): 'ab'
VARCHAR(n)	Variable-length string up to n characters	VARCHAR(2): 'ab', VARCHAR(5): 'abcd'
NCHAR & NVARCHAR	As above, but with Unicode support	NCHAR(2): 中文
TEXT	Variable-length string with no set limit	'a short sentence', 'a longer sentence'
DATE	Date in YYYY-MM-DD format	'2018-10-01', '2020-05-15'

Table 2: Examples of data types in SQL

SQLite Types

For more detailed information on SQLite types, visit the official documentation: <https://www.sqlite.org/datatype3.html>

- **Most SQL DBMSs** use a *static*, rigid typing system.
 - With static typing, a value's datatype is determined by the column in which the value is stored.
 - e.g. You **can't** store a string in a column defined as an integer.
- **SQLite**, on the other hand, uses a more general *dynamic* type system.
 - In SQLite, the datatype of a value is associated with the value itself, rather than strictly with the column's datatype.
 - e.g. You **can** store a string in a column defined as an integer, but SQLite will attempt to apply type affinity rules, converting the value if possible or storing it as text.

DATE and DATETIME

SQLite does **not** have dedicated **DATE** or **DATETIME** types. Dates and times are typically stored as **TEXT** (ISO-8601 format), **REAL** (Julian day numbers), or **INTEGER** (Unix time).

https://www.sqlite.org/lang_datefunc.html

SQLite defines **5 affinity types** that a column's datatype can be assigned to:

- **TEXT**: For storing text strings.
- **NUMERIC**: For numbers stored as integers or floating-point values.
- **INTEGER**: For storing whole numbers.
- **REAL**: For storing floating-point numbers.
- **BLOB**: For binary data, stored exactly as it was input.

Example: Module Table (1/2)

i Module Table Definition

The **Module** table contains information about university modules. Each module is identified by a unique 8-character code, along with a title and its associated credit value.

```
CREATE TABLE Module (  
    ...  
);
```

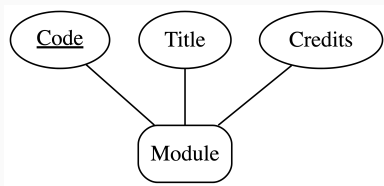


Figure 3: ER Diagram for the Module Table

Example: Module Table (2/2)

```
CREATE TABLE Module (  
    mCode CHAR(8) PRIMARY KEY,  
    mTitle VARCHAR(100) NOT NULL,  
    mCredits INTEGER NOT NULL DEFAULT 10  
);
```



The DEFAULT clause

The **DEFAULT** clause allows you to define a default value for a column. If no value is provided for that column when a new row is inserted, the default value will automatically be applied.

Relationships

Example: Student-Module-Enrolment

- Currently, we have two tables:
 - **Student**
 - **Module**
- We need to add a table, **Enrolment** to represent the relationship between **Student** and **Module**.
- The **Enrolment** table will have two columns:
 - **sID** – a foreign key that references the primary key in the **Student** table.
 - **mCode** – a foreign key that references the primary key in the **Module** table.

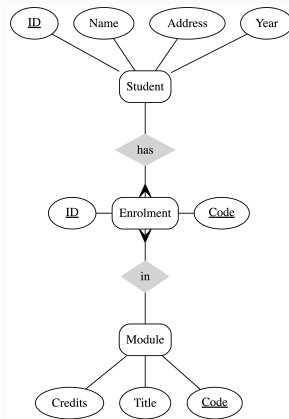


Figure 4: ERD for the Student, Module and Enrolment example



Foreign Keys must reference a **UNIQUE** (typically **PRIMARY KEY**) column

A foreign key must reference a **UNIQUE** column in the referenced table, otherwise foreign key constraints cannot be enforced.

- Foreign keys are used to create *relationships* between tables.
- **M:1** (Many-to-One) relationship: Represented by a foreign key in the *many* table.
- **M:M** (Many-to-Many) relationship: Split into two **1:M** (One-to-Many) relationships.
 - A separate table, called a *link* or *junction* table, is used to represent the relationship between the two tables.
- Why Foreign Keys are important:
 - **Relationship Building**: They allow us to link data between tables.
 - **Data Integrity**: Ensure that relationships between tables are accurate and consistent.
 - **Data Consistency**: Ensure that data updates are propagated to all related tables.

Example: Add Columns to Enrolment Table

```
-- This code is incomplete, please see the  
-- next slide for the completed version.
```

```
CREATE TABLE Enrolment (  
    sID INTEGER NOT NULL,  
    mCode CHAR(8) NOT NULL,  
    ...  
);
```

Example: Adding Foreign Keys

```
CREATE TABLE Enrolment (  
    sID INTEGER NOT NULL,  
    mCode CHAR(8) NOT NULL,  
    -- Composite Primary Key  
    PRIMARY KEY (sID, mCode),  
    -- Specify that sID is a foreign key  
    FOREIGN KEY (sID)  
        -- References the Student table  
        REFERENCES Student(sID),  
    FOREIGN KEY (mCode)  
        REFERENCES Module(mCode)  
);
```

- The **FOREIGN KEY** constraint ensures that values in the specified column(s) must match values in the referenced column(s).
- The **REFERENCES** keyword defines the table and column(s) that the foreign key points to.
- The referenced column(s) must either be a **PRIMARY KEY** or have a **UNIQUE** constraint.

Visualising Foreign Key Relationships

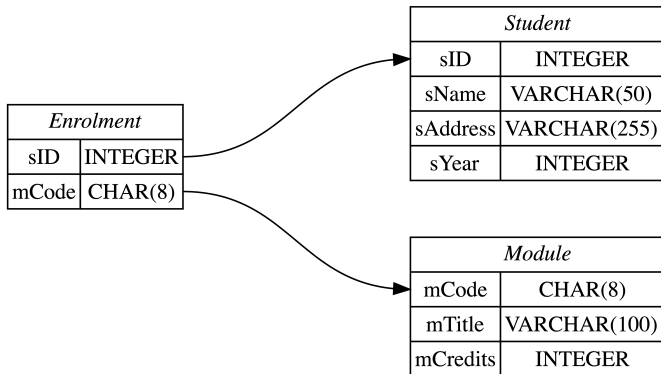


Figure 5: Visualisation of the foreign key relationships between the *Student*, *Module* and *Enrolment* tables.



A Common SQLite Gotcha: Foreign Key Constraints

By default, SQLite does not enforce foreign key constraints. To enable them, use the **PRAGMA** statement:

```
PRAGMA foreign_keys = ON;
```

PRAGMA is an SQLite-specific SQL command used to modify operational parameters, query the database's internal properties, or control the execution environment. Without this, foreign key constraints will not be applied, potentially leading to unexpected behavior.

- You can specify referential integrity constraints for each foreign key.
- These constraints are enforced during updates or deletions. Options include:
 - **RESTRICT**: Prevents updates or deletions that break referential integrity.
 - **CASCADE**: Automatically updates/deletes related rows.
 - **SET NULL**: Sets the foreign key to **NULL** in related rows.

Example: Add Referential Integrity Constraints

```
CREATE TABLE Enrolment (  
    sID INTEGER NOT NULL,  
    mCode CHAR(8) NOT NULL,  
    PRIMARY KEY (sID, mCode),  
    CONSTRAINT en_fk1  
        FOREIGN KEY (sID) REFERENCES Student(sID)  
        -- If a student is deleted, delete their enrolments  
        ON UPDATE CASCADE  
        ON DELETE CASCADE,  
    CONSTRAINT en_fk2  
        FOREIGN KEY (mCode) REFERENCES Module(mCode)  
        -- If a module is deleted, delete all associated enrolments  
        ON UPDATE CASCADE  
        ON DELETE CASCADE  
);
```

Deleting Tables using DROP



Practice Caution using **DROP**

Be **very careful** with this command. It will delete the table and all its data. There is no undo.

- Use the **DROP** keyword to delete tables:
 - `DROP TABLE [IF EXISTS] table-name;`
- Example:
 - `DROP TABLE Student;`
 - `DROP TABLE IF EXISTS Student;` (avoids errors if the table doesn't exist)
- **Foreign key constraints** may block deletion if another table references it:
 - Delete the referencing table first.
 - To enforce foreign key constraints in SQLite, enable them with:
 - `PRAGMA foreign_keys = ON;`

Reference Section

CREATE Table Definition

```
CREATE TABLE table-name (  
    col-name-1 col-def-1,  
    col-name-2 col-def-2,  
    ...  
    col-name-n col-def-n,  
    constraint-1,  
    ...  
    constraint-k  
);
```

- `table-name` is the name of the table to be created
- `col-name-n` is the name of the n-th column
- `col-def-n` is the definition of the n-th column
- `constraint-k` is the k-th constraint on the table

CREATE Column Definition



Non-Exhaustive List of Column Constraints

More information: https://www.sqlite.org/lang_createtable.html

```
col-name col-def  
[NULL | NOT NULL]  
[DEFAULT default_value]  
[NOT NULL | NULL]  
[AUTO_INCREMENT]  
[UNIQUE]  
[PRIMARY KEY]
```

- `col-name` is the name of the column
- `col-def` is the definition of the column
- `NULL` or `NOT NULL`: whether the column can contain `NULL` values
- `DEFAULT default_value`: specifies a default value for the column
- `AUTO_INCREMENT`: column is an auto-incrementing integer
- `UNIQUE`: must contain unique values
- `PRIMARY KEY`: column is a primary key

```
CONSTRAINT name
  FOREIGN KEY
    (col1, col2, ...)
  REFERENCES
    table-name
    (col1, col2, ...)
ON UPDATE ref_opt
ON DELETE ref_opt
```

- You need to provide:
 - A name for the constraint
 - The name of the column(s) in the referencing table
 - The name of the table being referenced
 - The name of the column(s) in the referenced table
 - The action to take when the referenced row is updated
 - The action to take when the referenced row is deleted
- `ref_opt` can be : `RESTRICT` | `CASCADE` | `SET NULL` | `SET DEFAULT`



SQLite dot commands

More information: <https://www.sqlite.org/cli.html>

- The SQLite Command Line Interface (CLI) has special commands dot commands .
- . commands control the behaviour of the CLI
- The most useful commands are:
 - **.help** - Display a list of commands
 - **.tables** - Display a list of tables
 - **.import** - Import data from a file into a table
 - **.read** - Execute commands from a file
 - **.schema** - Display the schema of a table
 - **.quit** - Exit the command line tool

Extra-Study Exercise: Pilot Qualification Database

💡 Problem Description

A pilot can be qualified to fly multiple aircraft, and an aircraft can be flown by many pilots. All pilots must have a name and age. All pilots begin with 1 year of experience (from training). All aircraft must have all attributes.

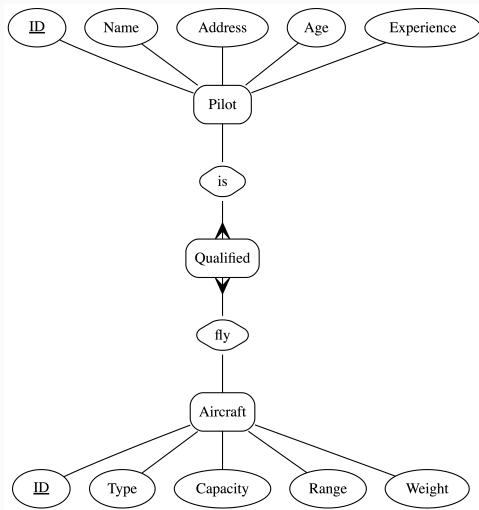


Figure 6: ERD for the Pilot Qualification example

Codd, E. F. (1970). A relational model of data for large shared data banks. *Communications of the ACM*, 13(6), 377–387.

Wikipedia. (2023). *SQL — Wikipedia, the free encyclopedia*. <https://en.wikipedia.org/wiki/SQL>.