Databases

PIC 40A, UCLA

Databases

A **database** is a structured collection of data that can be accessed in different ways. A **relational database** can be thought of almost like a spreadsheet where we may wish to view a list of employee salaries, view all the information on a given employee, etc.

Databases

Within a database, we have a series of **record**s (tuples of values); and each value is associated to a particular **field** (**attribute**). This set of data is called a **table.**

Name	**Employee ID**	**Salary**	**Department**
Alice Foo	123	98000	Marketing
Bob Bar	456	93500	Marketing
Cindy Baz	789	84220	Engineering
David Quuz	591	110000	Engineering
Joe Bruin	888	145000	Software

Example of record is the tuple (row): **Cindy Baz, 789, 84220, Engineering**.

The column headings are the fields, i.e., the properties we store such as name, ID, salary, and department.

Databases

From a database, we can select a **view**, giving us a subset of the data. For example, from the table on the previous slide, we could extract only the names and salaries of records with a department of marketing:

Name	**Salary**
Alice Foo	98000
Bob Bar	93500

SQL

Structured Query Language (SQL) is a standardized **query language** to work with databases. A query language is a programming language designed for databases.

SQL is standardized by the International Organization for Standardization (ISO). Different implementations exist that adopt SQL or subsets of its standards. While most SQL platforms are open source, the up-to-date standards and drafts must be purchased.

Here is a link to the SQL Standardization from 1996 that can be downloaded for free.

SQL

SQL is a very high level programming language, even higher than JavaScript or PHP. It is termed a **declarative language**, where we *tell the computer what we want done* rather than telling it how do do things.

If our table with employee data was called **employees** then extracting the view of names and salaries of people in the marketing department would amount to:

```
SELECT name, salary FROM employees WHERE
  department='Marketing';
```

We are practically writing English commands!

SQLite

SQLite is a lightweight SQL engine. The engine takes up little space and access to data is up to 3 times as fast as other SQL implementations. Its databases can handle a large volume of data (in the Terabytes) and frequent reads/writes.

Some of its drawbacks include not supporting concurrency, i.e., simultaneous reads/writes from different sources can corrupt the data and cause crashes; being less secure since data cannot be protected any more than ordinary files on the server; and not supporting all SQL specifications.

Another plus is that while not supporting all SQL specifications, it adds some of its own.

SQLite

SQLite is a **serverless** SQL engine. This means that its files can be read and accessed directly. Other SQL engines are often implemented as another process on the server so that database access is managed only through the server as an intermediary.

Data Types

SQLite supports the following data types:

- **NULL** - a value that is missing or unknown
- **INTEGER** - a signed integer type, up to 8 bytes
- **REAL** - a floating point type, 8 bytes
- **TEXT** - for text, has no maximum size
- **BLOB** - short for "binary large object", data will be stored exactly as entered, no maximum size

The SQL Standardization specifies that strings should be enclosed in single quotes. SQLite doesn’t impose it, but we may as well follow that recommendation for the sake of other engines.

Important Commands

The following are some important commands in SQLite:

CREATE: to create a new table
INSERT: to create a record and place it in a table
SELECT: to retrieve records
UPDATE: to modify records
DELETE: to delete records
ALTER: to modify table
DROP: to delete a table

Commands do not need to be in all caps, but this is the established convention.

Running SQlite

Sqlite can be run through the command line. On the PIC server, one can write:

```
sqlite3 file_name.db
```

in order to open an SQlite (version 3) session with a database called **file_name**. If the database doesn't exist yet, it will be created.

SQLite Syntax

All statements in SQLite terminate with a semicolon. Pressing [ENTER] does not run a statement until a semicolon is typed.

There are a few exceptions. Some commands begin with a **.** and do not require semicolons:

- **.help** - see help documentation
- **.read** - read an SQL file
- **.quit** or **.exit** - quit the session
- **.tables** - list all tables
- many others...

CREATE

With **CREATE**, we can make a new table within a database. If we wish to create a table, the syntax is:

CREATE TABLE name_of_table (
field1 affinity1,
field2 affinity2,
...
);

where **name_of_table** is the table we are creating; the **field**s are the names of the fields in the table, and the **affinity**s are the corresponding **affinity** of that field.

CREATE

The **affinity** is just a recommendation to the engine as to what sort of data should be stored there. Given data of a different type than the specified affinity, the engine will try to make a conversion. If it is successful, the converted value is inserted; otherwise, the mismatched type is inserted.

For example, if a column is supposed to store INTEGER, inserting the string ’123’ would result in 123 being stored, but the string ’hello world’ would be stored as TEXT type with value ’hello world’.

CREATE

SQLite3 supports the following affinities:

- **NUMERIC** - integer (if possible) or floating point
- **INTEGER**
- **REAL**
- **TEXT**
- **BLOB**

So **NULL** is not an affinity. And **NUMERIC** is an affinity but not a type.

CREATE

To avoid creating a table that already exists, which would result in an error, we can instead opt for the creation step:

```
CREATE TABLE IF NOT EXISTS name_of_table (
field1 affinity1,
field2 affinity2,
...
);
```

This is more robust and necessary in the case of running SQLite for a website.

CREATE

Below we open an SQLite3 session for the company database and make a table ready to store employee information:

```
sqlite3 company.db

CREATE TABLE IF NOT EXISTS employees(
name TEXT,
id INTEGER,
salary REAL,
department TEXT
);
```

INSERT

The **INSERT** command allows us to insert a record into a table. The syntax is:

INSERT INTO table_name (field1, field2, ...) VALUES (value1, value2, ...);

The parentheses **()** are important. The value for **field1** is set to **value1**, etc.

If we do not give all fields a value, the record is inserted and its value for that field is NULL.

INSERT

Continuing our employees example:

```
INSERT INTO employees (name, id, salary, department)
  VALUES ('Alice Foo', 123, 98000, 'Marketing');

INSERT INTO employees (name, id, salary, department)
  VALUES ('Bob Bar', 456, 93500, 'Marketing');
```

INSERT

It is possible to omit the column names when we are inserting an entire row. However, this is not particularly robust because if the order of the variables gets mixed up, it can lead to bugs and data corruption. *Purely for illustration:*

```
INSERT INTO employees VALUES ('Cindy Baz', 789, 84220,
 'Engineering');
```

INSERT

We can insert multiple records at once, as well, by using a comma between inserted rows.

```
INSERT INTO employees (name, id, salary, department)
VALUES ('David Quuz', 591, 110000, 'Engineering'),
('Joe Bruin', 888, 145000, 'Software');
```

SELECT

The **SELECT** statement is a way to make SQL display things. It is very much like a print statement. For example,

```
SELECT 9*8, 3-4, 12/2;
```

will display:

```
72|-1|6
```

SELECT

Of course, we are more concerned with using this keyword to get information from a database... Here are some common structures:

SELECT field1, field2, ... FROM table_name;

SELECT field1, field2, ... FROM table_name WHERE condition;

SELECT field1, field2, ... FROM table_name WHERE condition1 AND condition2;

SELECT field1, field2, ... FROM table_name WHERE condition1 OR condition2;

The first lists the values for the given fields found in the table.
The second does the same, but only includes the records where a condition is satisfied.
The third and fourth show how logical AND and OR conditions can be used.

SELECT

In SQLite, we have:

- **<** for less than
- **<=** for less than or equal
- **=** or **==** for equal
- **<>** or **!=** for not equal
- **>=** for greater than or equal
- **>** for greater than

And to condition upon values of a column, we reference that column name. For example, **WHERE id > 400** requires that a record has id above 400 to be considered true.

SELECT

Continuing our example:

```
SELECT name FROM employees;
```

```
Alice Foo
Bob Bar
Cindy Baz
David Quuz
Joe Bruin
```

SELECT

```
SELECT name, department, id FROM employees WHERE
 id > 700 OR department='Engineering';
```

```
Cindy Baz|Engineering|789
David Quuz|Engineering|591
Joe Bruin|Software|888
```

SELECT

Remark: there is also a **SELECT *** syntax to select all fields from a table. While it can be useful for debugging purposes to see what is in a table, it is considered poor practice to use it in production code. The reason is that columns of a database can change and * is unaware of this.

Here's how it would look (for debugging only):

```
SELECT * FROM employees;
```

Then the entire table is printed.

UPDATE

The **UPDATE** statement allows us to modify records. A typical pattern is:

UPDATE table_name SET field1=value1, field2=value2, ... WHERE condition;

UPDATE

We can give Alice a raise and change her department.

```
UPDATE employees SET salary = 122321.86,
  department='Engineering' WHERE name='Alice Foo';
```

Recall that **salary** was **REAL** so the value matches the affinity.

DELETE

To delete records, we use **DELETE**:

DELETE FROM table_name WHERE condition;

DELETE

Perhaps Bob leaves the company:

```
DELETE FROM employees WHERE name= 'Bob Bar';
```

LIMIT

At times, we may wish to only get a limited number of records from a query. This can be done through with **LIMIT**:

```
SELECT name FROM employees LIMIT 3;
```

```
Alice Foo
Cindy Baz
David Quuz
```

In this example, we only see the first three records.

AS

Through the **AS** command, we can create new tables from other tables/views. For example, we could create a new table to only store the name and email address of employees in a company who work in Engineering:

```
CREATE TABLE engineering AS
  SELECT name, id FROM employees WHERE
  department = 'Engineering';

SELECT name, id FROM engineering;
```

```
Alice Foo|123
Cincy Baz|789
David Quuz|591
```

PRAGMA

To get a listing of fields, we write:

PRAGMA table_info(name_of_table);

For example, with

```
PRAGMA table_info( engineerng );
```

We may obtain

```
0|name|TEXT|0||0
1|id|INTEGER|0||0
```

where the first few entries are the field index, name, and affinity.

PHP and SQLite

We will now look at running SQLite through PHP. It is actually quite easy. Here is an example. We first connect to a database.

```
try{ // attempt to establish connection

  $mydb = new SQLite3('company.db'); // opens or creates the database
}
catch(Exception $ex){ // may throw
  echo $ex->getMessage();
}
```

When we attempt to connect to a database (or make a new one), PHP can throw an exception. Exceptions are handled almost the same in PHP as they are in C++, hence the familiar **try** and **catch** blocks.

PHP **Exception** objects can store information about what went wrong that can be accessed with **getMessage**.

PHP and SQLite

Now we make a table.

```
// what we would write directly in the command line is in quotes

$statement = 'CREATE TABLE IF NOT EXISTS employees(name TEXT, id INTEGER, salary REAL, department TEXT);';

$run = $mydb->query($statement); // run the command
```

We simply put the SQLite commands into a string and run the commands through the **query** function.

The **query** function returns an **SQLite3Result** object if it is successful and otherwise returns **false**.

PHP and SQLite

We need to be careful about using the quotes for strings as SQLite strings need to be within quotes.

```
1  // enclose in double quotes to make single quotes easier
2
3  $statement = "INSERT INTO employees (name, id, salary, department) VALUES ('Alice Foo', 123,
         98000, 'Marketing');";
4  $run = $mydb->query($statement); // run the command
5
6  // feeding variables into queries
7  $name = 'Bob Bar';
8  $id = 456;
9  $salary = 93500;
10 $department = 'Marketing';
11
12 $statement = "INSERT INTO employees (name, id, salary, department) VALUES ('$name', $id,
        $salary, '$department');";
13 $run = mydb->query($statement);
```

In the statement created on line 12, we needed the single quotes to indicate an SQL string. Because the variables were enclosed in double quotes, the PHP engine renders them as the value they store.

PHP and SQLite

When we do a SELECT query, we expect to receive a collection of data. That data is stored in the **SQLite3Result** object. There is a member function **fetchArray** that returns a record (row of a result) as an associative array with key-value pairs being the field index name - value and field name - value pairs in the table. When no more records are found, it returns **false**.

Recall that PHP treats an object as **true**.

```php
$statement = 'SELECT name, id FROM employees;';
$run = $mydb->query($statement);

if($run){ // so no errors in the query
  while($row = $run->fetchArray()){ // while still a row to parse
    echo $row['name'], '--', $row['id'], '<br/>'; // print all the data
  }
}
```

```
Alice Foo - 123
Bob Bar - 456
```

PHP and SQLite

The condition **while($row = $run->fetchArray())** may look strange but all it is doing is creating/modifying a variable **$row** to be the array that is fetched.

As in other programming languages, assignment returns a value. In this case, after being assigned/created, **$row** is returned, which can be coerced to **true** when it is a non-empty array and otherwise it is **false**. Thus, the **while** only runs when there is an array to parse.

The array can be indexed from 0 or by field name. Thus, **$row['name']** means the same as **$row[0]**.

PHP and SQLite

After using a database, we should **close** the connection. We can then open again with **open**.

```
$mydb->close();

// later
$mydb->open('other.db');
```

ALTER

With **ALTER**, we can rename tables or add fields with syntax such as:

ALTER TABLE table_name RENAME TO new_name;

ALTER TABLE table_name ADD COLUMN field affinity;

ALTER

Here we will add an email field for the employees.

```
ALTER TABLE employees ADD COLUMN email TEXT;
```

With the work above, now all records have an **email** field with value **NULL** (until we **UPDATE**).

DROP

A table can be deleted with **DROP**. The syntax is:

DROP TABLE IF EXISTS table_name;

Remark: with **DROP**, we cannot delete a field. To do that, we must create a new, temporary table, copy only the desired values over to it, destroy the original, and rename the temporary to the original name.

DROP

Let's say we decided to remove the **email** field from **employees**. It is long and painful. The **--** are comments.

```
BEGIN TRANSACTION; -- need this to all happen at once
CREATE TABLE IF NOT EXISTS
  employee_temp_table(name TEXT, id INTEGER,
  salary REAL, department TEXT);
INSERT INTO employee_temp_table (name, id, salary,
  department) SELECT name, id, salary, department
  FROM employees;
DROP TABLE IF EXISTS employees;
ALTER TABLE employee_temp_table RENAME TO employees;
COMMIT; -- end the process
```

DROP

We saw the single line comments are prefixed by **--**. Multiline comments begin with /* and end with */.

Transactions are processes that are supposed to be "atomic", i.e. they represent one single step that all has to take place before other steps/processes. If an error occurs, the entire process is unwound. We enclose the atomic process in the **BEGIN TRANSACTION** and **COMMIT** statements. The notion of atomic processes is helpful when multiple threads/processes could be accessing the database simultaneously.

The process entailed: making a new table with only the fields we want, selecting the relevant data from the old table, dropping the old table, and renaming the new table.

DROP

The data returned from **SELECT** can feed directly into an **INSERT** statement.

For robustness we use the **IF EXISTS** and **IF NOT EXISTS** statements to check if a table existed or did not yet exist.

ORDER BY

We can add sorting instructions to a **SELECT** query with **ORDER BY**. The keywords **ASC** and **DESC** indicate ascending/descending order. Sorting is done based on parsing commands from left to right.

SELECT field1, field2, ... FROM table_name ORDER BY fieldx direction, fieldy direction, ...;
where **fieldx**, **fieldy**, ... represent fields and **direction** is either **ASC** or **DESC**.

ORDER BY

With the employees table:

```
SELECT name, department FROM employees ORDER BY
  department ASC, name DESC;
```

```
David Quuz|Engineering
Cindy Baz|Engineering
Alice Foo|Engineering
Joe Bruin|Software
```

Remember that Bob left the company so he is not here and we moved Alice to Engineering.

NOT NULL

It can sometimes be useful to ensure that some fields are always given, i.e., not missing for any records. By specifying a field as **NOT NULL** this can be done.

```
CREATE TABLE IF NOT EXISTS contact_list(
  name TEXT NOT NULL, phone INTEGER NOT NULL,
   email TEXT );
INSERT INTO contact_list (name, phone)
  VALUES ( 'Stephanie Balan', 5551234567); -- okay

-- INSERT INTO contact_list(phone, email)
  -- VALUES (5550009999, 'person@email.com');
  -- ERROR! name NULL
```

Reading Files

In an SQLite3 session, we can run commands through reading from a file. For example, suppose that **art.sql** stores the following:

```
CREATE TABLE IF NOT EXISTS art
  (artist TEXT NOT NULL, piece TEXT NOT NULL,
  year INTEGER);
INSERT INTO art (artist, piece, year)
  VALUES ('A. Y. Jackson', 'Red Maple', 1914);
```

After opening a database, we could run:

```
.read art.sql
```

and the table would be created with the Jackson art piece added.

Remark: sometimes reading too many times can cause strange permissions errors. Deleting the concerning tables seems to be the best remedy.

PRIMARY KEY

Sometimes we want to preserve uniqueness of sorts in a table. For example, not having two people with the same ID. The syntax is:

CREATE TABLE IF NOT EXISTS table_name (field1 affinity1, ..., primary_field primary_affinity PRIMARY KEY, fieldx affinityx, ...);

PRIMARY KEY

We can have an error generated if two people are assigned the same ID:

```
CREATE TABLE IF NOT EXISTS employees2(name TEXT,
  id INTEGER PRIMARY KEY, salary REAL,
  department TEXT);

INSERT INTO employees2 VALUES ('Ellen', 100,
  48010.50, 'Advertising');

-- will produce an errror! id of 100 already used
INSERT INTO employees2 VALUES ('Frank', 100,
  47998.00, 'IT');
```

PRIMARY KEY

We can also have an automatically incremented index as a primary key. We can specify it when inserting a record but if we do not, it will default to one higher than the current largest index. We need to specify **AUTOINCREMENT**.

```
CREATE TABLE IF NOT EXISTS to_do_list(index
  INTEGER PRIMARY KEY AUTOINCREMENT, task TEXT);

INSERT INTO to_do_list (task) VALUES ('plan to take
  over the world'), ('take over world');

SELECT (index, task) FROM to_do_list;

1|plan to take over the world
2|take over the world
```

INNER JOIN

Through SQLite, we can also join tables in various fashions. One such case is the **INNER JOIN** command where we can extract data from multiple tables where some values line up.

For fields that could be ambiguous between tables, we need to specify **table_name.field_name** to disambiguate.

We look at an example...

INNER JOIN I

```
CREATE TABLE IF NOT EXISTS class( name TEXT NOT NULL,
   uid INTEGER NOT NULL);
CREATE TABLE IF NOT EXISTS grades(
  uid INTEGER NOT NULL,
  percent REAL NOT NULL);

INSERT INTO class (name, uid) VALUES
  ('Becky Wong', 111111 ), ('Henry Ives', 999999),
  ('Marshall Edgecombe', 777777),
  ('Luz Rodriguez', 444444);

INSERT INTO grades( uid, percent) VALUES
  (111111, 94.7), (444444, 90.1), (999999, 83.2),
  (777777, 80.2);

CREATE TABLE IF NOT EXISTS combined AS
```

INNER JOIN II

```
  SELECT name, class.uid, percent
  FROM class
    INNER JOIN grades ON class.uid = grades.uid
    ORDER BY percent DESC;

SELECT name, uid, percent FROM combined;
```

INNER JOIN

```
Becky Wong|111111|94.7
Luz Rodriguez|444444|90.1
Henry Ives|999999|83.2
Marshall Edgecombe|111111|80.2
```

In this example, we joined the two tables based on where the university IDs matched up, creating a new table with the records sorted in descending order by overall percent.

Statistics

With SQLite3, we can also compute things like mean-values of fields. For example with

```
SELECT avg( percent ) FROM combined
  WHERE percent > 90;
```

we obtain:

```
92.4
```

SQL Injections

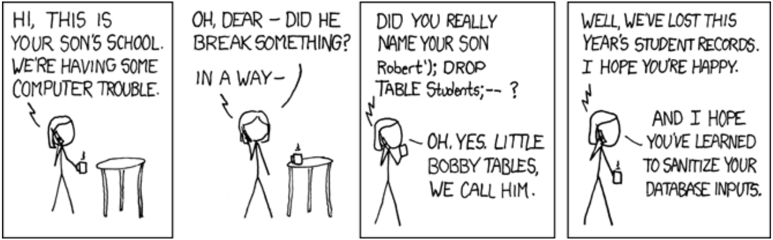
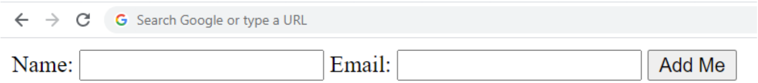


Figure: Courtesy of XKCD titled “Exploits of a Mom”.

SQL Injections

Malicious users can, through their keyboard inputs, do malicious things by **SQL Injections.** Consider this webpage where a user can enter their username and email address that is written to a database.

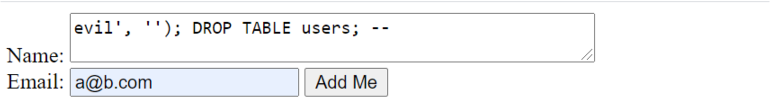


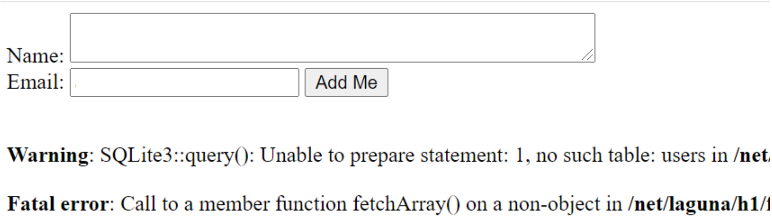
SQL Injections

```
#!/usr/local/bin/php
<?php
  $db = new SQLite3('users.db');
  $db->query("CREATE TABLE IF NOT EXISTS users (name TEXT, email TEXT);");
?>
<!DOCTYPE html>
<html>
<head>
  <title>Title</title>
</head>
<body>
  <form method="POST" action="<?php echo $_SERVER['PHP_SELF']; ?> " >
    <label for="name">Name: </label> <textarea name="name" id="name" cols="50"></textarea>
    <br/>
    <label for="em">Email: </label> <input type="email" name="em" id="em" cols="50" />
    <input type="submit" value="Add Me" name="add" />
  </form>
  <p>
    <?php
      if( isset($_POST['add']) ){ // if submitted, show all users
         $cmd = "INSERT INTO users (name, email) VALUES ( '" . $_POST['name'] . "', '" .
              $_POST['em'] . "');";
         $db->query($cmd);
         $test = "SELECT name, email FROM users;";
         $res = $db->query($test);
         while( $arr = $res->fetchArray() ){ // each record
           var_dump($arr);
           echo '<br/>';
         }
      }
      $db->close();
    ?>
  </p>
</body>
</html>
```

SQL Injections

In the example, malicious inputs were used to delete the database!





When parsed, the name ends up deleting the database!

SQL Injections

The preceding example is an example of a **primary SQL injection** where the data the user enters directly wreaks havoc on the database. A **secondary SQL injection** is one where the data the user stores, when later retrieved, causes problems.

Consider a database that stores usernames and (hashed) passwords. Suppose a hacker is after the user **bobfoo**. The hacker could register for an account with name **bobfoo', ' '); - -**. If the the hacker then updates their password to **sucks_to_be_bob**, a query string of the form

```
UPDATE TABLE users SET password = 'sucks_to_be_bob'
  WHERE user = 'bobfoo';
  -- AND password = hacker_password;
```

would be generated thus changing **bobfoo**'s password to **sucks_to_be_bob**!!!

SQL Injections

A **prepared statement** is a way to guard against these attacks. It involves preparing an SQL statement ahead of time and then substituting in key expressions with specific types/requirements. The issue is fixed when the lines 21-22 are replaced by:

```
// :name and :email will get replaced -- quotes will be added!
$ins = $db->prepare( "INSERT INTO users (name, email) VALUES ( :name, :em );" );

// assign values for :name and :em, both of type TEXT
$ins->bindValue(':name', $_POST['name'], SQLITE3_TEXT);
$ins->bindValue(':em', $_POST['em'], SQLITE3_TEXT);

// execute the database command safely
$ins->execute();
```

prepare makes a prepared statement; **bindValue** assigns the placeholder values; and **execute** runs the prepared statement.

SQL Injections

Besides **SQLITE3_TEXT** for text, there are also **SQLITE3_INTEGER**, **SQLITE3_FLOAT**, **SQLITE3_BLOB**, and **SQLITE3_NULL**.

Warning: when **SQLITE3_TEXT** is used, the prepared statement **automatically adds single quotes around the entry**!

Summary

- SQL is a **declarative language** use to interface with databases.
- SQLite can be run without a server. It is fast and can manage large amounts of data.
- The language has 5 data types, namely **NULL**, **INTEGER**, **REAL**, **TEXT**, and **BLOB**; along with 5 affinities, namely **NUMERIC**, **INTEGER**, **REAL**, **TEXT**, and **BLOB**.
- Standard SQL commands do not terminate until a semi-colon.
- Some common commands include **CREATE**, **INSERT**, **SELECT**, **UPDATE**, **DELETE**, **ALTER**, **DROP**, and **ORDER BY**.
- Through **SQLite3** objects, PHP can create/write to/access data from databases.

Summary

- The **query** member function of **SQLite3** queries can be run with either **SQLite3Result** objects being returned if successful or **false** being returned if there has been an error.
- Through the **fetchArray** member function of **SQLite3Result**, we obtain doubly-indexed array of records.
- Deleting fields from a table is not an easy process and needs to be done through **TRANSACTION**s.
- SQL supports making fields into the **PRIMARY KEY** and also allows **AUTOINCREMENT**.
- An **SQL injection** is an attack that can be carried out on a database to corrupt or compromise the data.
- **Prepared statements** can protect against these attacks.

Exercises I

1. What are the 5 data types of SQLite3? What are the 5 affinities? What is the difference between a data type and an affinity?
2. Write a series of SQLite3 commands to:
   - Create a table (if it doesn't exist) with name 'students' to store the names and UID's of all students in a class
   - Insert "Joe Bruin" of UID 123456789 into the class.
   - Extract the names and UIDs of all students (assume more) whose UID is between 100000000 and 200000000 inclusive.
3. Write a web interface for the database in the previous exercise.
   - There should be one page, **add.html** where a user can enter a student name and id numbers in a web form and, upon submission, these values are added to a database.
   - There should be another page **listing.html** where the user can enter a lower/upper value for the UID (or leave either/both fields blank) into a webform and upon pressing a "List" button, they see all students with their names and IDs displayed on the page whose UIDs are at least as big as the lower value (if specified) and no larger than the upper value (if specified).