CMSC 23500 — Introduction to Database Systems
Discussion Session #3

April 14, 2008

1 SQLite

In the first part of this course, we will be using the SQLite relational DBMS (`http://sqlite.org/`. This is a light-weight DBMS that will allow us to experiment with SQL and to get a feel for how a database works, without having to go through the more complex interactions involved in using heavier systems like MySQL, PostreSQL, Oracle, etc.

One of SQLite's main advantages is that it stores a database in a single file which can be easily copied and moved around, and accessed directly through a command-line interface or simple APIs. Larger DBMSs also store their data in files, but they are rarely (if ever) manipulated directly by the user/programmer. Instead, they must be accessed through a database server, which is typically not easy to set up. On the other hand, SQLite is simple to install and requires no configuration, so you can start toying and tinkering with it right away, and using an existing database is as easy as making a copy of a database file. More details about SQLite's main features can be found at `http://www.sqlite.org/different.html`.

Of course, SQLite does have its drawbacks. Most notably, it does not scale well to high-concurrency applications (e.g., high-traffic websites) or large datasets. Additionally, it does not support all the features of SQL, although this will not be a problem for the simple queries we will be seeing in this lab.

2 Using SQLite

Starting SQLite from one of the CS machines is as simple as running the `sqlite3` command:

```
$ sqlite3
SQLite version 3.3.8
Enter ".help" for instructions
sqlite>
```

This starts an SQLite shell where we can type in SQL statements. Of course, there are other ways to access an SQLite database other than typing in raw SQL statements into a shell. For example, there are GUIs (SQLiteman is a popular one, although not installed on the CS machines; see `http://sqliteman.com`) and APIs for several programming languages (see `http://www.sqlite.org/cvstrac/wiki?p=SqliteWrappers` for a complete list). However, for now we will keep the focus on interacting with the database through SQL commands.

Since we started the SQLite shell without accessing an existing database, the only types of statements we could run at this point would be DDL statements (e.g., to create a new table). In this lab you are provided with an example database (you can download it from the course website), and we will focus on using DML statements to access and manipulate the contents of that database. To start the SQLite shell, and attach it to the database, just run the following:

```
$ sqlite3 planet_express.db
```

Besides SQL, this shell also provides several administration commands, which all begin with a dot. Two commands we will find useful are `.tables` and `.schema`. The first one will show a list of all the tables in the database:

```
sqlite> .tables
Client         Has_Clearance  Planet
Employee       Package        Shipment
```

The `.schema` command shows the DDL command that was used to create a table:

```
sqlite> .schema Planet
CREATE TABLE Planet (
        PlanetID INTEGER PRIMARY KEY NOT NULL,
        Name TEXT NOT NULL,
        Coordinates REAL NOT NULL
);
```

Running `.schema` without any parameter will dump the entire schema for the database. Note that you will see several statements related to "TRIGGER"s. You can ignore them for now. They have been added because SQLite does not enforce referential integrity automatically, so we must rely on triggers, a common mechanism found in relational databases, to enforce referential integrity.

Now, let's try to run an SQL query that returns the entire contents of the `Planet` table. Type in the following:

```
sqlite> SELECT * FROM Planet;
```

You should see the following output:

```
1|Omicron Persei 8|89475345.3545
2|Decapod X|65498463216.3466
3|Mars|32435021.65468
4|Omega III|98432121.5464
5|Tarantulon VI|849842198.354654
6|Cannibalon|654321987.21654
7|DogDoo VII|65498721354.688
8|Nintenduu 64|6543219894.1654
9|Amazonia|65432135979.6547
```

By default, SQLite makes no attempt to pretty-print the results of a query (this is good to process the output of a query with another program, but not that good if it's going to be read by a human). You can make the output look nicer by running the following administration commands:

```
sqlite> .mode column
sqlite> .headers on
```

If you rerun the previous query, you should now see the following:

```
PlanetID    Name              Coordinates
----------  ----------------  -------------
1           Omicron Persei 8  89475345.3545
2           Decapod X         65498463216.3
3           Mars              32435021.6546
4           Omega III         98432121.5464
5           Tarantulon VI     849842198.354
6           Cannibalon        654321987.216
7           DogDoo VII        65498721354.6
8           Nintenduu 64      6543219894.16
9           Amazonia          65432135979.6
```

Finally, note that you can break a query into as many lines as you want. The end of a query is always delimited by a semicolon. For example:

```
sqlite> SELECT *
    ..> FROM Planet;
```

3 The Planet Express database

Before running more elaborate SQL queries, take a look at Figure 1, which shows the relational schema of the example database. This schema models the data from the Planet Express delivery company, focusing on keeping track of shipments. In particular, the company has several employees, each of which can be the manager for

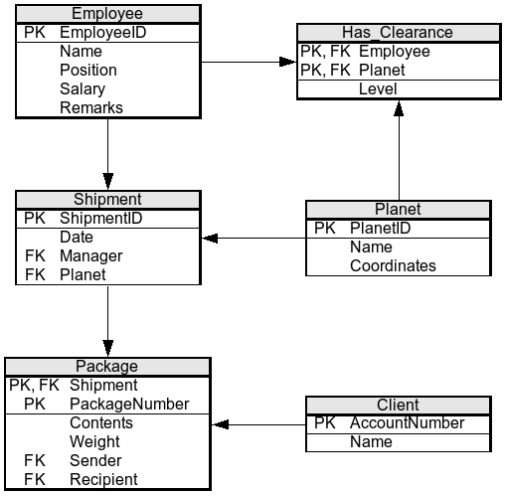

Figure 1: Relational schema for the Planet Express database

zero or more shipments. A shipment includes several packages, all delivered to the same planet. If a shipment has been carried out, the shipment date is noted in the database. Otherwise, its value will be NULL (indicating this is a pending shipment). Employees, furthermore, can be cleared (by the pertinent authorities) to land in zero or more planets. Each clearance has a specific level, from 1 to 4. Packages in each shipment will have a code that is unique within that shipment, and the database will keep track of who the sender and recipient of the package are (both of which will have an account number with the company).

Make sure you understand the diagram. More specifically, look at the primary keys and foreign keys. Do they make sense? Are they consistent with the description provided above? Can you identify what foreign keys are the result of a 1..N relationship, and which ones are a result of an N..M relationship? Is there any table that originated from a weak entity?

4 A couple sample queries

You will be seeing SQL in detail in this week's lectures. So, take the following queries with a healthy dose of salt. Their purpose is for you to become familiar with running SQL queries in SQLite and to demonstrate how SQL has a simple syntax providing similar functionality to the relational algebra. For example, the following query shows how you can do a projection (instead of selecting all the columns from a table):

```
sqlite> SELECT EmployeeID, Name, Position FROM Employee;
```

You should see the following output:

```
EmployeeID  Name            Position
----------  --------------  ------------
1           Phillip J. Fry  Delivery boy
2           Turanga Leela   Captain
3           Bender Bending  Robot
4           Hubert J. Farn  CEO
5           John A. Zoidbe  Physician
6           Amy Wong        Intern
7           Hermes Conrad   Bureaucrat
8           Scruffy Scruff  Janitor
```

SQL also allows you to specify selection conditions:

```
sqlite> SELECT EmployeeID, Name, Position FROM Employee
    ..> WHERE Salary >= 10000;
```

The above query should return the following:

```
EmployeeID  Name          Position
----------  ------------  ----------
2           Turanga Leela  Captain
4           Hubert J. Far  CEO
7           Hermes Conrad  Bureaucrat
```

Next, let's see what happens if we try to violate the primary key integrity of a table. The following statement adds a new tuple to the `Planet` table, setting the primary key to 4. If you look at the results of our first query, which returned all the contents of the `Planet` table, you'll see that there is already a tuple with that primary key.

```
sqlite> INSERT INTO Planet(PlanetID, Name, Coordinates)
    ..> VALUES (4, ''Jupiter'', 1839102.5);
```

This statement should result in the following error: `SQL error: PRIMARY KEY must be unique`

Finally, let's see what happens if we violate referential integrity. Remember that a foreign key in table A must have the value of the primary key in the existing tuples of table B (where, furthermore A and B can be the same table). A foreign key can also be NULL, if allowed by the schema. The following statement creates a new shipment that refers to a planet that does not exist in the `Planet` table:

```
sqlite> INSERT INTO Shipment(ShipmentID, Date, Manager, Planet)
    ..> VALUES (10, ''11/04/3004'', 1, 50);
```

This statement should fail, presenting the following error: `SQL error: insert on table ‘‘Shipment’’ violates foreign key constraint ‘‘fki_Shipment_Planet_-Planet_PlanetID’’`

5 Exercises

As you can see, the SQL syntax is pretty simple. Give it a whirl, and see if you can do the following (the expected result is shown so you can verify if your query was correct):

1. Select the packages from shipment 3.

```
 Shipment    PackageNumber  Contents    Weight      Sender
----------  -------------  ----------  ----------  ----------
3           1              Undeclared  15.0        3
3           2              Undeclared  3.0         5
3           3              Undeclared  7.0         2

 Recipient
----------
4
1
3
```

2. Select the packages from shipment 3, with a weight larger than 10.

```
 Shipment    PackageNumber  Contents    Weight      Sender
----------  -------------  ----------  ----------  ----------
3           1              Undeclared  15.0        3

 Recipient
----------
4
```

3. Select the contents and weight of all the packages.

```
Contents    Weight
----------  ----------
Undeclared  1.5
Undeclared  10.0
A bucket o  2.0
Undeclared  15.0
Undeclared  3.0
Undeclared  7.0
Undeclared  5.0
Undeclared  27.0
Undeclared  100.0
```

4. Select all the pending shipments.

```
 ShipmentID  Date        Manager     Planet
----------  ----------  ----------  ----------
3                       2           3
4                       2           4
5                       7           5
```

5. Create a new shipment that, unlike the query shown above, does not violate referential integrity. Verify that the insertion was successful by viewing the contents of the `Shipment` table.