19 Overview of Relational Databases

19.1 Lecture

1. Databases are used to store information in a structured manner. This means that there are lots of rules that you can set up which the database will enforce. While this can be frustrating and seemingly pedantic at times, it does mean that when you use your data, you will have hard guarantees about its contents and format; this makes your applications much more robust.
2. We will be working with 'relational database' technology. In a relational model, all data are stored in 'tables'. These tables have well defined columns, which are called 'fields'. Each row in a table is called a 'record'. The key element of the relational model is that records in different tables can be associated with one another, typically based on common field values.
   (a) Consider these three tables ...

Table 1

Nation	Continent
USA	North America
Italy	Europe
Spain	Europe
France	Europe

Table 2

Artist	Nation
Dante	Italy
Davinci	Italy
Degas	France
Michelangelo	Italy
Miro	Spain
Pei	USA

Table 3

Artist	Discipline
Dante	author
Davinci	architect
Davinci	painter
Davinci	sculptor
Degas	painter
Michelangelo	architect
Michelangelo	painter
Michelangelo	sculptor
Miro	painter
Pei	architect

   (b) You can determine the continent that Dante comes from by ...
      i. Find the record in Table II where the value of the Artist field is 'Dante'.
      ii. Note the value of the Nation field ('Italy') in this table.
      iii. Find the record in Table I where the value of the Nation field is 'Italy'.
      iv. Note the value of the Continent ('Europe') field in this record.
   (c) Relational databases do things like this very quickly and efficiently.
3. Some common relational databases that you may have heard of include Oracle, IBMs DB2, Microsoft SQL Server, MySQL, and PostgreSQL. These are large relational database systems that are appropriate for enterprise applications. They support many users at the same time, have complex security models, and lots of advanced features. Because of this, it is quite involved to set them up, and consequently you need to do quite a bit of work before you can perform even the simplest actions (such as creating your first table and putting some data in it).

20 Installing Open Source Software Under Linux

20.1 Lecture

1. One of the great advantages of working in the Linux ecosystem is that there is a wealth of free, open source software available (there is also quite a bit of such software available for Windows and Mac OS X). Quality varies from really poor to best-of-breed, enterprise class; often the hardest part of using open source software is finding the right package or application to meet your needs.
2. To facilitate our exploration of relational databases, we will install a lightweight relational database tool called sqlite3. We will do this by downloading the source code for the program and compiling it ourselves.
   (a) You can often find 'pre-compiled binaries' for many programs; these are usually easier to install and get running. However, compiling the programs yourself (aka: 'installing from source') often leads to better compatibility.
   (b) In practice, only one of us would have to build sqlite3 and everyone could use that copy of the program. However, for instructional purposes, we'll all install our own copies.
   (c) Installation procedures for open source software vary widely the best thing to do is read the instructions (often found in a file called README or INSTALL that comes with the distribution).
   (d) The installation of sqlite3 is typical of many packages.
3. The quality of the install process and documentation is usually a very good indicator of the quality of the package. If you can't get the package to build properly on a normal system without too much fuss, you may want to reconsider using that particular package.

20.2 Exercise

1. Download the source code tarball.
   (a) Open your laptop's web browser and navigate to `http://www.sqlite.org/`
   (b) Go to the 'Download' page.
   (c) Right click (or control-click on a Mac) on the second link in the 'Source Code' section to select the file for the command line program (`sqlite-autoconf-3240000.tar.gz`) and choose 'Copy Link Location'. This will paste the URL for this file onto the clipboard. Note that you don't want to download this file onto your laptop; you want it on the Linux computer.
   (d) Go back to the terminal and create a directory called `downloads` in your home directory. This is a common convention for storing files that have been downloaded from the Internet. **`cd`** into this directory.
   (e) Now download the file from the internet by entering the **`wget`** command below; note that you don't have to type the URL, just paste it using the terminal's paste command (mouse right-click if you are using putty).
      i. `https://sqlite.org/2018/sqlite-autoconf-3240000.tar.gz`
2. Unpack the distribution tarball (you do remember how to do that, right?).
   (a) Usually you want to do this in a temporary directory.
   (b) Have a look at the README and CONFIGURE files in the distribution
   (c) The size of the sqlite3 package is quite small. It is not uncommon to see hundreds or thousands of files in larger packages.
3. To build sqlite3 enter the following commands:
   (a) **`./configure --prefix=$HOME/sqlite3`**
   (b) **`make`**
   (c) **`make install`**
4. Typically, a good open source software package will install for all users on a system with the com-

mands: **./configure ; make ; make install**. However, the last step will try to put the software in directories that you are not allowed to write to. The **--prefix** option to the **./configure** command tells the installer to install this program in your home directory.

   (a) Have a look in your **${HOME}/sqlite3** directory. What do you think you need to add to your PATH?

5. Phew! To test the installation, enter the following command: **sqlite3**

   (a) To quit sqlite3, enter **.quit** (remember, when learning a new program, the first thing to learn is how to get out of it).

6. ... and there was much rejoicing!

21 Your First Database

21.1 Lecture

1. Now that we have a working database program, let's create our first database. Because we are a very cultured group, we will create a database of artists.
   (a) **`sqlite3 artists.db`**
2. We'll now create our first table, and add a few records into it.
   (a)
   ```
   sqlite> CREATE TABLE nations (
       ...> nation VARCHAR(80),
       ...> continent VARCHAR(80)
       ...> );
   ```
   (b)
   ```
   sqlite> INSERT INTO nations (nation, continent)
   VALUES ('USA', 'North America');
   ```
   (c)
   ```
   sqlite> INSERT INTO nations (nation, continent)
   VALUES ('Italy', 'Europe');
   ```
   (d)
   ```
   sqlite> INSERT INTO nations (nation, continent)
   VALUES ('Spain', 'Europe');
   ```
   (e)
   ```
   sqlite> INSERT INTO nations (nation, continent)
   VALUES ('France', 'Europe');
   ```
3. To see the tables defined in a database, use the **`.schema`** command.
4. Next, well check that our data were added as expected.
   (a)
   ```
   sqlite> SELECT * FROM nations;
   USA|North America
   Italy|Europe
   Spain|Europe
   France|Europe
   ```
5. The commands we have been entering at the sqlite¿ prompt are SQL commands. As you probably noticed, SQL commands end with a semicolon; if you dont enter the semicolon, you get the chance to continue the SQL command on the next line, as was done with the **`CREATE TABLE`** command.
   (a) The keywords in SQL commands are not case sensitive; however, it is a common convention to write the SQL keywords in uppercase.
   (b) Different database systems have different rules for the case sensitivity of database objects (e.g., table names, field names). It is usually best to assume that they are case preserving, but not case sensitive. Also, avoid special characters and whitespace in database object names.
6. Most relational databases use the same SQL language. When you get to more advanced features, there are some subtle differences.
7. Let's take a closer look at the **`CREATE TABLE`** command. We asked for a table with two fields (columns), and we specified that each column should contain values of 'type' **`VARCHAR(80)`**. This tells sqlite that it will be storing character data (or 'strings') in these fields. Furthermore, they are suggested to be limited to 80 characters.

21.2 Exercise

1. Add a record for South Korea to the nations table in your database.
2. CHALLENGE: Try to add a record for China, but use its full, official name: "The People's Republic of China". Hint: read question 14 of the FAQ for sqlite.

22 Importing Data

22.1 Lecture

1. Adding records to a database one at a time using SQL insert statements is less than fun, and is not the typical mechanism for populating a table. If you have data in text files, a much more efficient way to bring the data into a database is to import it.
2. When we started sqlite, the program kindly told us to 'Enter **`.help`** for instructions'. This is a hint that we should . . . you guessed it . . .
3. Note the **`.import`**, **`.mode`**, and **`.separator`** commands. Can you guess how one would import a tab delimited file?
4. When importing data, it is not uncommon to have to try a few options before you get it right. When fighting with your database, you may need to change your table definitions, and/or clear all of the records from a table before you attempt to repopulate it.
   (a) Remember, you need to create a table before you can import records into it.
   (b) To change a table definition, you need to get rid of the old table definition and create a new one. To get rid of an old definition, use the SQL command: **`DROP TABLE`** *`tablename`* **`;`**
   (c) It is often helpful to keep long SQL commands in a separate file that you can edit and re-execute (and refer to later). The **`.read`** command facilitates this.
   (d) To clear all the records from a table (but not change its definition), use the SQL command: **`DELETE FROM`** *`tablename`* **`;`**
5. The set of all table definitions in a database is called its schema.

22.2 Exercise

Note: You may find it helpful to develop a single file that contains all of the commands for this exercise.

1. Create tables to hold the data shown in Tables II and III in the Overview of Relational Databases section above.
2. Import data into these tables from the files in ~**`unixinst/artists`** directory.
   (a) Inspect these files manually to discover their format.
3. To check that you have imported the correct number of records, you can run a **`SQL SELECT query: SELECT COUNT(*) FROM tablename;`**
   (a) With which UNIX command can you compare this result?
4. Here is a file that achieves all of the above steps (as well as some of the previous commands).
   (a) Note the use of the **`IF EXISTS`** clauses in the **`DROP TABLE`** commands.
   (b) Note the use of absolute path names to refer to files. Why do you think this was done?

```
DROP TABLE IF EXISTS nations;
CREATE TABLE nations (
  nation VARCHAR(80),
  continent VARCHAR(80)
);

INSERT INTO nations (nation, continent)
  VALUES ('USA', 'North America');
INSERT INTO nations (nation, continent)
  VALUES ('Italy', 'Europe');
INSERT INTO nations (nation, continent)
  VALUES ('Spain', 'Europe');
INSERT INTO nations (nation, continent)
  VALUES ('France', 'Europe');

DROP TABLE IF EXISTS artists;
CREATE TABLE artists (
  artist VARCHAR(80),
  nation VARCHAR(80)
);

.mode csv
.import /home/unixinst/artists/artist_nation artists

DROP TABLE IF EXISTS artist_discipline;
CREATE TABLE artist_discipline (
  artist VARCHAR(80),
  discipline VARCHAR(80)
);

.mode tabs
.import /home/unixinst/artists/artist_discipline artist_discipline
```

23 Querying Databases Part I (Single Tables)

23.1 Lecture

1. The SQL **SELECT** command is used to query data from a database. This is a very complex command, and we will only be scratching the surface here and in the next section.
2. We have already seen the most basic **SELECT** command.
   (a) `SELECT * FROM artists;`
      i. In this command, the `*` means 'all fields'. If you want to see just the list of artists, you can enumerate the field names you are interested in.
   (b) `SELECT artist FROM artists;`
3. You can ask for only those records that meet certain criteria.
   (a) `SELECT artist FROM artists WHERE nation = 'Italy';`
4. Note that some queries will contain duplicate results.
   (a)
   ```
   SELECT artist FROM artist_discipline
   WHERE (discipline = 'sculptor' OR discipline = 'painter');
   ```
   (b) In this case, we see some duplicates because we are only asking for the artist field, and we are seeing results for each record in the underlying table that matches our search criteria.
5. Duplicates in a result set can be eliminated with a **DISTINCT** clause.
   (a)
   ```
   SELECT DISTINCT artist FROM artist_discipline
   WHERE (discipline = 'sculptor' OR discipline = 'painter');
   ```
6. By default, there is no intrinsic order to a result set. You will often see records in the order in which they were added to the table, but this is not guaranteed. You can run the same query twice and get the results back in a different order.
   (a) `SELECT nation FROM nations;`
7. To ask the database to sort results, use the **ORDER BY** clause.
   (a) `SELECT continent, nation FROM nations ORDER BY nation;`
8. You can use sqlite commands within a pipeline. On the **sqlite3** command line, place the (single) SQL command you want to run after the database file name. The output of the query will go to stdout.
   (a)
   ```
   sqlite3 artists.db "select artist from artist_discipline \
   where discipline = 'painter';" | egrep '[io]$'
   ```


24 Querying Databases Part II (Joining Tables)

24.1 Lecture

1. Although you can get a lot of mileage out of querying individual tables, the real power of relational databases derives from their ability to efficiently join data from multiple tables.
2. A join of two tables can be understood as first constructing a virtual table that contains all possible combinations of all of the records from each source table, and then whittling down this virtual table by applying the field selectors and SQL **`WHERE`** conditions that are part of the query (see the color figures for a visual explanation).
   (a) Databases don't actually do this; they are quite smart and take many shortcuts. However, the results you get are identical to those you would have gotten had the database actually constructed, and then filtered, such a virtual table, so this is a useful mental model to have.
   (b) All of the SQL clauses you have learned about so far (**`WHERE`**, **`ORDER BY`** and **`DISTINCT`**) can be used in queries that involve joins.
   (c) You can even do triple joins (a query across three tables).
3. Sometimes it is difficult (although often usually possible) to answer your question in one SQL statement. In this case, you may want to create temporary tables to hold intermediate results.
   (a) You can create temporary tables with the **`CREATE TEMPORARY TABLE`** command.
   (b) Temporary tables are automatically dropped when you exit **`sqlite`**. This prevents the accumulation of clutter.
   (c) In the bigger database systems, temporary tables are only visible per database session, so your temporary tables won't interfere with the work of others, even when you are working off of the same database.
   (d) Both the **`CREATE TABLE`** and **`INSERT INTO`** SQL statements allow you to use a **`SELECT`** statement to specify data to be added.
4. In an ideal database, you only store raw data. Any questions you have relating to the data are computed at the time the question is asked using a SQL query. Storing intermediate results in additional tables is problematic because if the underlying raw data are updated, your intermediate tables become out of date. Most often, the best way to solve this problem is to avoid it.

24.2 Exercise

1. List all European architects, in alphabetical order.
2. List all European sculptors whose names contain the letter 'm'.
   (a) HINT: Find the description of the **`LIKE`** operator at `https://sqlite.org/lang_expr.html`
3. How many European artists are there (write a SQL query that returns this number; don't list them and count them yourself).

25 Schema Design and Normalization

25.1 Lecture

1. Designing a database schema is as much art as science. One important goal of schema design is called normalization; this can be summed up as: "the database should be designed such that it is impossible for the data in it to be inconsistent". The primary consequence of this is that you store each raw fact only once.
2. This schema for this table is not normalized because it records the fact that Italy is in Europe multiple times.

Table 4: Bad Table

Artist	Nation	Continent
Dante	Italy	Europe
Davinci	Italy	Europe
Degas	France	Europe
Michelangelo	Italy	Europe
Miro	Spain	Europe
Pei	USA	North America

   (a) If one of the records were changed to (incorrectly) indicate that Italy was in Africa, the data in the database would be internally inconsistent. Experience has taught us that it is better to be consistently wrong than to be inconsistent (it is easier to fix errors in the former case).
   (b) CHALLENGE: Consider the case of Turkey, which is both in Europe and Asia. Can you think of a schema able to model this?
3. There are no comment fields that SQL can understand for noting exceptions to your schema. If there is even one case that doesn't fit into your schema, you've got a problem and need to modify your schema.
4. Don't just assume that every file you import should correspond to its own table in the schema.
   (a) For example, being provided with a file listing Italian artists, and another file listing American artists, does not imply you should have two such tables. You can always recover the original files' contents with an appropriate query.
   (b) A good acid test is to check if you would lose any information if you were to rename a file to a nonsense name. If so, then you probably should add a field to hold that information. All records imported from that file would have the same value. Information should be stored in records, not in table names.
   (c) If two tables have the same fields (i.e., they have the same logical meanings), you might consider merging them into one table.

26 A Practical Example

26.1 Exercise (as a group)

1. The ~**unixinst/nimblegen** directory contains some real data from epigenetic microarray experiments. Explore the data files and see if you can make any sense of them (this is how it works in the 'real world').
2. For efficiency, database files should always be on a local hard disk. On our system, this means placing the file in the **/tmp** filesystem. Make your own subdirectory there.
3. Design and implement a schema to hold this data.
   (a) How many tables do you need?
   (b) Is there any redundant information that should be removed to achieve a normalized database design?
   (c) It is sometimes a good idea to create indices on fields that are heavily used in searches and joins.
      i. Use the **CREATE INDEX** SQL command to do this.
      ii. Selection of fields to index is often made empirically by testing how long it takes to execute particular queries.
4. Import the data into the database.
   (a) When importing de-normalized data, it is often a good idea to check for internal consistency.
   (b) What would you do if you found any inconsistencies?
5. Develop a query to print out the minimum, average, and maximum spot intensity for each probe set (i.e., each SEQ_ID).
   (a) HINT: Use the **GROUP BY** clause in the **SELECT** command.