

8

SQL

Contents

8.1	Foundations of SQL	120
8.1.1	Summary	120
8.1.2	Applications of SQL.	122
8.2	Defining Tables	124
8.3	Data Manipulation and Queries	125
8.4	Relations	130
8.5	Practical Examples	133

Goals

- Understanding the applications of SQL
- Defining simple tables with SQL
- Manipulating data and forming queries

Prerequisites

- Knowledge of the basic Linux commands
- Basic text editing skills

8.1 Foundations of SQL

8.1.1 Summary

The “Structured Query Language” (SQL) is a standard language for defining, querying, and manipulating relational databases. Relational databases store data records (also called “tuples” in tech-speak) in “tables”. You can visualise a table by imagining a large sheet of paper which is divided into rows and columns. The rows are the records stored in the table, and the columns describe the properties of the records (Table 8.1). The database makes it convenient to retrieve those tuples that match specific criteria (like the family names of all persons who command a starship named “USS Enterprise”).

What makes the whole thing interesting is a concept called “normalisation”. If you take a closer look at the initial example, you will notice that the names of some persons, ships, and films occur multiple times. This is not desirable, because modifications to these data would have to be applied in multiple places inside the database. There is a danger of missing one place and introducing inconsistent data. Instead, the database is “normalised”: We know that the “James T. Kirk” in the first two tuples is actually one and the same person, but the “USS Enterprise” in the first three tuples and the one in the fourth tuple are different vessels¹. So we can split our table into three, one each for the people, ships, and films. This can be seen in Table 8.2. Please note the following remarked:

- The original table has morphed into a new table named “Person”, which no longer contains the columns holding the ship names and the films. Instead, the name of a ship is given using a “foreign key” that refers to a tuple of the “Ship” table. This foreign key us to express the fact that both Willard Decker and James T. Kirk commanded the “old” USS Enterprise, while Jean-Luc Picard commanded the “new” one. This is called a “1 : n relationship”, because the same ship may have multiple commanding officers during the course of its existence.
- The same person may appear in different films and the same film may feature multiple people from the “Person” table. This is why

¹Well, “trekkies” like us know it, but *not* knowing it may not actually make you a bad person.

First Name	Surname	Starship	Film
James T.	Kirk	USS Enterprise	Star Trek
James T.	Kirk	USS Enterprise	Star Trek: Generations
Willard	Decker	USS Enterprise	Star Trek
Jean-Luc	Picard	USS Enterprise	Star Trek: Generations
Han	Solo	Millennium Falcon	Star Wars 4
Han	Solo	Millennium Falcon	Star Wars 5
Wilhuff	Tarkin	Death Star	Star Wars 4
Mal	Reynolds	Serenity	Serenity

Figure 8.1: A database table: Famous spaceship commanders from films

Person	First Name	Surname	Ship	PersonFilm	Person	Film
1	James T.	Kirk	1	1	1	1
2	Willard	Decker	1	2	1	2
3	Jean-Luc	Picard	2	3	2	1
4	Han	Solo	3	3	3	2
5	Wilhuff	Tarkin	4	4	4	3
6	Mal	Reynolds	5	5	4	4
				6	5	3
				7	6	5

Ship	Name	Film	Title	Year	Budget (Million \$)
1	USS Enterprise	1	Star Trek	1979	46
2	USS Enterprise	2	Star Trek: Generations	1994	35
3	Millennium Falcon	3	Star Wars 4	1977	11
4	Death Star	4	Star Wars 5	1980	33
5	Serenity	5	Serenity	2005	39

Figure 8.2: Famous spaceship commanders from films (normalised)

the relationship between people and films cannot be expressed through a simple foreign key in the “Person” table. (What we have here is called an “ $m : n$ relationship”). Instead, we introduce yet another table whose tuples represent propositions of the form “person x appears in film y ”.

- We added a few columns to the “Film” table just to make things a bit more interesting.



For simplicity’s sake, we ignore the fact that one of our people might command several different spaceships during the course of their career.

At first glance, this “data model” may appear somewhat more complicated, but it does store every piece of information in only one place, which makes it a lot easier to keep things under control in “real life”.




Relational databases were originally proposed by Edgar F. Codd in 1970. Even today they form the backbone of computer-based data processing. Relational databases can reflect the object-oriented data structures of modern software only to a limited degree, but unlike fancy new approaches like “object-oriented databases” they have the clear advantage of being based on sound mathematical theory (“relational algebra”), as well as being amenable to reasonably efficient implementation.





The first version of SQL (then still called SEQUEL) was developed in the early 1970s by Donald D. Chamberlin and Raymond F. Boyce. It formed the basis of IBM’s first relational database system, System R. SQL was standardised for the first time in 1986, but development continued afterwards. The current version of the standard is

ISO 9075:2008 (popularly know as ISO SQL:2008). Unsurprisingly, it was ratified in July 2008.

 The official pronunciation of SQL is “S-Q-L”. Occasionally people will also pronounce it like the word “sequel”.

Exercises


 **8.1** [!2] How would you add some additional ship crew members to the data model defined in this section?

 **8.2** [2] Where in the data model would you place the director of a film?


8.1.2 Applications of SQL

As we said before, SQL is an essential part of today’s commercial data processing—a whole industry thrives on implementing and supporting products implementing relational database systems. Here are some SQL-based relational database products available for Linux:

MySQL and PostgreSQL These two packages are probably the first ones that come to mind when thinking of the terms “Linux” and “SQL”. Both are freely available and quite popular and form a solid basis for common, not overly complex applications, for example in the area of the World Wide Web.

 As usual in the open-source community, there are vigorous “holy wars” between the advocates of both packages. PostgreSQL disciples decry the fact that MySQL does not implement all of the SQL standard, while MySQL proponents argue that the parts that MySQL *does* implement are entirely adequate while the speed of MySQL makes it easy to forgo the remainder. We shall not give a recommendation either way here; both products are freely available and can be evaluated on their merits as required.

Oracle, Sybase, DB2, and friends There is a whole bunch of commercially implemented and supported database systems for “mission-critical” applications that (also) run on Linux. These systems offer all the features of implementations based on Windows or traditional Unix systems and can be used without hesitation for all kinds of large-scale database applications.

 While you can run MySQL and PostgreSQL on essentially any Linux system, the commercial database manufacturers usually limit their official support to a number of specific platforms, typically the “enterprise” distributions from companies like Red Hat and Novell/SUSE, on which they “certify” their products—this means that the manufacturer tests the product thoroughly on the platform in question, proclaims that it works, and is subsequently prepared to help paying customers that run exactly that platform if they experience problems.

Naturally you are free to get Oracle and friends to run on other Linux distributions than the officially certified ones, and chances are good that that will work (it's not as if Linuxes were *that* different from one another). However, you'd be on your own in case of trouble, which does cast some doubt on why you would want to use an expensive commercial database system in the first place—since for most applications you could resort to MySQL and PostgreSQL, too.



Incidentally, it is not a big problem to obtain “commercial” support for PostgreSQL and MySQL, too (at commercial rates).

SQLite While the other packages typically provide a “database server” in a separate process to which application programs connect over the network, SQLite is linked directly to application programs as a library, is usable without configuration, and reads and writes local files. SQLite supports most of the SQL92 standard, including features like transactions and triggers which may be problematic even with MySQL. SQLite is suitable for use on low-memory devices such as MP3 players or as a data format for application programs.

Throughout the rest of this chapter we will use SQLite to illustrate SQL.



On Debian GNU/Linux or Ubuntu, you can easily install SQLite by using one of the following commands:



```
# aptitude install sqlite3           for root
$ sudo aptitude install sqlite3      for other users
$ sudo apt-get install sqlite3       on Ubuntu
```

Make sure to get `sqlite3`—there is also `sqlite`, which is an obsolete version which is only still offered for compatibility.



On OpenSUSE 11, SQLite (3) is part of the default installation, just like on Fedora 10. So you do not need to do anything special to try the examples in this chapter—everything you need is already installed.



Exercises



8.3 [2] Under which circumstances would you use a freely available SQL database product like MySQL or PostgreSQL for a web site? Does your evaluation depend on the nature of the web site, i.e. whether it is a hobby project or part of a mission-critical task?



8.4 [2] The authors of SQLite recommend SQLite to store application program data (think of the tables of a spread sheet or the configuration data of a web browser). Which advantages and disadvantages of the approach can you think of?

```

CREATE TABLE person (
  id          INTEGER PRIMARY KEY,
  firstname   VARCHAR(20),
  surname     VARCHAR(20),
  ship_id     INTEGER
);

CREATE TABLE film (
  id          INTEGER PRIMARY KEY,
  title       VARCHAR(40),
  year       INTEGER,
  budget     INTEGER
);

CREATE TABLE ship (
  id          INTEGER PRIMARY KEY,
  name       VARCHAR(20)
);

CREATE TABLE personfilm (
  id          INTEGER PRIMARY KEY,
  person_id  INTEGER,
  film_id    INTEGER
);

```

Figure 8.3: The complete schema of our sample database

8.2 Defining Tables

Before you can fill an SQL database with data, you have to specify the names of the individual tables, the names of the columns, and the nature of the values to be stored in a column. SQL supports a large variety of data types like “string” or “integer” that you can resort to when defining the columns of a table. All the table definitions of a database together are incidentally called a “database schema”.



A discussion of the full SQL language standard is beyond the scope of this document. With table definition in particular there are also large differences between the various SQL databases. We limit our discussion to the absolute minimum, also because *creating* tables is not part of the LPI-102 exam.

In SQL syntax, a definition of the “Person” table from our example might look like

```

CREATE TABLE person (
  id          INTEGER PRIMARY KEY,
  firstname   VARCHAR(20),
  surname     VARCHAR(20),
  ship_id     INTEGER
);

```

INTEGER and VARCHAR(20) denote the SQL data types “integer” and “string of up to 20 characters”. The “PRIMARY KEY” clause declares the id column the “Rand[Primary Key]primary key”. This means that the database ensures that any value in this column occurs only once, and the values here can serve as “foreign keys” in tuples from other tables (in the case of “person”, for example, the table joining people and films).



It is a common convention to give foreign keys the name of the table they “point to”, with a suffix of `_id`, so `ship_id` is a foreign key that refers to the `ship` table.



If you are somewhat familiar with SQL, you may object to our defining the foreign key, `ship_id`, as a mere INTEGER. Please allow us this simple view of things for today.



Incidentally, SQL does not distinguish between uppercase and lowercase characters. We follow the common convention of putting the names of tables and columns in lowercase and everything that is proper SQL in uppercase, but you can basically suit yourself. However you will do yourself and us a favour by being consistent with yourself.

Figure 8.3 shows the complete SQL schema of our sample database. If the schema is stored in the `commanders-schema.sql` file, you could initialise the actual database using SQLite as follows:

```
$ sqlite3 comm.db <commanders-schema.sql
```

This command stores the database in the `comm.db` file. SQLite always takes the name of a database file as a parameter; if the database file exists already, it will be opened, otherwise it will be created.

When you run `sqlite3` without redirecting standard input, you get an interactive session:

```
$ sqlite3 comm.db
SQLite version 3.5.9
Enter ".help" for instructions
sqlite> .tables
film      person   personfilm  ship
sqlite> .schema ship
CREATE TABLE ship (
  id      INTEGER PRIMARY KEY,
  name    VARCHAR(20),
);
sqlite> _
```



SQLite features various “metacommands” whose names all begin with a dot—in the example you can see `.tables`, which lists the tables in a database, and `.schema`, which lets you inspect the database schema. There are many more, though; `.help` gets you the complete list.

Exercises



8.5 [!2] Create an SQLite database based on the spaceship commander database schema given in this section.



8.6 [3] Implement the extensions from Exercise 8.1 and Exercise 8.2 as an SQL schema.

8.3 Data Manipulation and Queries

SQL does not only support defining database schemas, but also inserting, modifying, querying, and deleting data (tuples). All of this is subject to the current database schema. For example, you might add a few ships and films to our database:

```

sqlite> INSERT INTO ship VALUES (1, 'USS Enterprise');
sqlite> INSERT INTO ship VALUES (2, 'USS Enterprise');
sqlite> INSERT INTO film VALUES (1, 'Star Trek', 1979, 46);
sqlite> INSERT INTO film VALUES (2, 'Star Trek: Generations',
...> 1994, 35);

```



Note that we specify explicit values for the primary keys here. In a larger database this is somewhat tedious, since you would have to figure out the right value for every new tuple—it is much more convenient to have the database itself insert the correct value, and most SQL database can in fact do this. With SQLite, the primary key must be declared as an “INTEGER PRIMARY KEY”, and you must pass the “magical” value NULL instead of an explicit value:

```
sqlite> INSERT INTO film VALUES (NULL, 'Star Wars 4', 1977, 11);
```

People and tuples specifying the person-film relation can be entered likewise:

```

sqlite> INSERT INTO person VALUES (1, 'James T.', 'Kirk', 1);
sqlite> INSERT INTO person VALUES (2, 'Willard', 'Decker', 1);
sqlite> INSERT INTO personfilm VALUES (NULL, 1, 1);
sqlite> INSERT INTO personfilm VALUES (NULL, 1, 2);
sqlite> INSERT INTO personfilm VALUES (NULL, 2, 1);

```

Note how we specify the primary keys of the tuples in the corresponding tables for the foreign keys `ship_id` in the `person` table and `person_id` and `film_id` in the `personfilm` table.



If you know a bit about programming, this may make you feel somewhat queasy. After all, nobody guarantees that there actually *is* a tuple with the corresponding primary key in the “other table”². This is not a fundamental problem with SQL but rather one with our simplistic examples—“good” SQL databases (not SQLite, and MySQL not always) support a concept called “referential integrity” which helps solve exactly this problem. It makes it possible to specify in the schema that `ship_id` is a foreign key to the `ship` table, and the database will then ensure that the values for `ship_id` remain reasonable. Referential integrity incorporates other nice properties, too; in our simple example you yourself would have to take care, when you remove the James T. Kirk tuple from the `person` table, to also remove the tuples from `personfilm` that connect James T. Kirk to films. With a database supporting referential integrity, this could happen automatically.

referential integrity

With the data from Table 8.2 in our database we can now look at a few queries. We can obtain all tuples from a table like this:

```

all tuples  sqlite> SELECT * FROM ship;
           1|USS Enterprise
           2|USS Enterprise
           3|Millennium Falcon
           4|Death Star
           5|Serenity

```

²Unless you are a C programmer, that is; in that case there is nothing wicked about this at all.

The asterisk (“*”) implies “all columns”. If you want to limit your selection to specific columns, you have to enumerate them:

specific columns

```
sqlite> SELECT firstname, surname FROM person;
James T.|Kirk
Willard|Decker
Jean-Luc|Picard
Han|Solo
Wilhuff|Tarkin
Mal|Reynolds
```

You are not restricted to retrieving column values exactly the way they are stored in the database, but can form “expressions” based on them. The following example lists the full names of the ship commanders without the ugly vertical bar. The “||” operator concatenates two strings.

Expressions

```
sqlite> SELECT surname || ', ' || firstname FROM person;
Kirk, James T.
Decker, Willard
Picard, Jean-Luc
Solo, Han
Tarkin, Wilhuff
Reynolds, Mal
```

Of course you can do calculations, too:

```
sqlite> SELECT title, budget * 0.755 FROM film;
Star Trek|34.73
Star Trek: Generations|26.425
Star Wars 4|8.305
Star Wars 5|24.915
Serenity|29.445
```

(Whether it makes a lot of sense to convert 1977 U.S. dollars to euros at the January 2009 exchange rate is a different question, though.)

“Aggregate functions” make it possible to apply operations like sums and averages to particular columns of all tuples. For instance, you calculate the number and the average budget of all films in our database (disregarding inflation) as follows:

Aggregate functions

```
sqlite> SELECT COUNT(budget), AVG(budget) FROM film;
5|32.8
```

Of course you only get a single tuple as the result.



The “COUNT(budget)” may surprise you a little, but it stands for “the number of all tuples in a table whose budget column actually contains a value”. It might be possible for the budget of a film to be unknown—“Star Trek”, for example, is a borderline specimen—and in this case you could enter the NULL value there (not to be confused with “0” for a film which didn’t cost anything to produce). Such films will then simply be skipped when the aggregate function is calculated. If you want to know the number of all films, no matter whether their budget is known or not, you can say “COUNT(*)”.

An interesting feature, especially when dealing with aggregate functions, is “grouping”. Assume we’re interested in the average budget of

grouping

the films produced in a decade, for all decades in the database. We could use something like

```
sqlite> SELECT year/10, AVG(budget) FROM film GROUP BY year/10;
197|28.5
198|33.0
199|35.0
200|39.0
```

(This may not be the greatest possible output format, but it does what it is supposed to.) The “GROUP BY” clause specifies that all tuples for which year/10 gives the same result are to be considered together, and the column specifications then refer to all the tuples in one such group. This means that AVG(budget) no longer calculates the average of *all* tuples, but only that of the tuples in the same group.

column names



So far the names of the output columns derived from the column names of the input tuples. SQL does allow you to request your own names for output tuples. Especially with more complex queries this can make things clearer:

```
sqlite> SELECT year/10 AS decade, AVG(budget)
...> FROM film GROUP BY decade;
```

is rather less tedious to read than the original.



How your output actually looks depends mostly on your database system. By default, SQLite is fairly simple-minded, which may be mostly due to the fact that, in the spirit of the “Unix toolchest”, it tries to produce output that is easily processed by other programs and free of superfluous chatter. For interactive use, though, you can select more convenient output formats:

```
sqlite> .headers on
sqlite> .mode tabs
sqlite> SELECT year/10 AS decade, AVG(budget)
...> FROM film GROUP BY decade
decade avg(budget)
197    28.5
198    33.0
199    35.0
200    39.0
```

Here the individual columns are separated by tabs. With “.mode column” you can obtain a format where you can assign explicit column widths using .width:

```
sqlite> .mode column
sqlite> .width 10 12
sqlite> SELECT year/10 AS decade, AVG(budget)
...> FROM film GROUP BY decade
decade      avg(budget)
-----
197         28.5
198         33.0
```

```

199      35.0
200      39.0

```

Frequently you do not want to work on all tuples from a table, but only a subset matching specific criteria. You can do this with SELECT, too. Here, selecting tuples for example, is the list of all films in our database that were produced since 1980:

```

sqlite> SELECT title, year FROM film WHERE year >= 1980;
Star Trek: Generations|1994
Star Wars 5|1980
Serenity|2005

```

You can also sort the output:

```

sqlite> SELECT title, year FROM film WHERE year >= 1980
...> ORDER BY year ASC;
Star Wars 5|1980
Star Trek: Generations|1994
Serenity|2005

```

“ASC” here means “ascending”, the opposite would be “DESC”:

```

sqlite> SELECT title FROM film ORDER BY budget DESC;
Star Trek
Serenity
Star Trek: Generations
Star Wars 5
Star Wars 4

```

The WHERE clauses of queries may contain SELECT]other SELECT com- Sub-SELECTs
mands as long as these deliver something that fits the selection expression
in question. Here is the list of all films with an above-average budget:

```

sqlite> SELECT title, year FROM film
...> WHERE budget > (SELECT AVG(budget) FROM film);
Star Trek|1979
Star Trek: Generations|1994
Star Wars 5|1980
Serenity|2005

```

You can modify tuples by means of the UPDATE command:

modifying tuples

```

sqlite> UPDATE person SET firstname='James Tiberius' WHERE id=1;
sqlite> SELECT firstname, surname FROM person WHERE id=1;
James Tiberius|Kirk

```

The WHERE clause is extremely important in this case so changes apply to specific tuples. One slip and the disaster is perfect:

```

sqlite> UPDATE person SET firstname='James Tiberius';
sqlite> SELECT firstname || ' ' || surname FROM person;
James Tiberius Kirk
James Tiberius Decker
James Tiberius Picard
James Tiberius Solo
James Tiberius Tarkin
James Tiberius Reynolds

```

But of course you can put this to profitable use:

```
sqlite> UPDATE film SET budget=budget * 0.755;    Convert to euros
```

deleting tuples Finally, you can use the DELETE FROM command to delete tuples from a table. The WHERE warning applies here, too:

```
sqlite> DELETE FROM person WHERE surname='Tarkin';
```

You can delete all tuples from a table using

```
sqlite> DELETE FROM person;    Kids, don't try this at home
```



Remember our remark above concerning “referential integrity”. Depending on the mojo of your database system, you may have to ensure by yourself that tuples containing foreign keys to a tuple will disappear along with that tuple.

Exercises



8.7 [1] Insert all tuples of Table 8.2 into the SQLite database described in Exercise 8.5. If you are into science fiction films, feel free to extend the database a bit. (For example, we are big fans of “Galaxy Quest”.)



8.8 [2] Give an SQL command that lists all films in our sample database that were produced before 1985 and had a budget of less than 40 million dollars.

8.4 Relations

SQL queries get really interesting when you combine multiple tables. Really interesting SQL queries combine multiple tables. For example, you might be interested in a list of all spaceship commanders together with the names of their ships (the tuples in person only contain the primary keys of the ships):

```
sqlite> SELECT * FROM person, ship
...>   WHERE person.ship=ship.id;
1|James T.|Kirk|1|1|USS Enterprise
2|Willard|Decker|1|1|USS Enterprise
3|Jean-Luc|Picard|2|2|USS Enterprise
4|Han|Solo|3|3|Millennium Falcon
5|Wilhuff|Tarkin|4|4|Death Star
6|Mal|Reynolds|5|5|Serenity
```

That’s a bit thick, isn’t it? But let’s take it step by step:

- The secret to our success is, once again, the WHERE clause. It puts the foreign key of the person table in relation (Eek, the R-word!) to the primary key of the ship table and thus causes the corresponding tuples to be matched.
- The output looks a bit messy because each tuple of ship is simply appended to the matching tuple of person. In fact we do not need both ship_id from person and id from ship, if the next thing we output is the ship name, anyway. Something like

```
sqlite> SELECT firstname, surname, name FROM <<<<<<
James T.|Kirk|USS Enterprise
<<<<<<
```

would be completely adequate. However, this only works because all the columns have distinct names, and SQLite can infer which table each name refers to. If the surname column in person was simply called name, there would be a conflict with the name column of ship.

The most common method for resolving name conflicts and abbreviating long SQL commands at the same time is using aliases:

aliases

```
sqlite> SELECT * FROM person p, ship s WHERE p.ship_id=s.id;
```

This is equivalent to the original example except that, for this command, we gave the person table the alias p and the ship table the alias s. This did make the WHERE clause that much easier to read.



Aliases can be used in the column lists as well, as in

```
sqlite> SELECT p.firstname, p.surname, s.name <<<<<<
```

This also lets you handle name collisions between the columns of different tables:

```
sqlite> SELECT firstname, p.name, s.name <<<<<<
```

The example we showed for joining two tables works but is to be enjoyed with some caution (see also Exercise 8.9). When two tables are joined in this manner, the database system first constructs the Cartesian product of the tables in question and then throws out all resulting tuples that do not match the WHERE condition. This means that what happens is roughly this:

```

Condition: The fourth and fifth columns must match
1|James T.|Kirk|1|1|USS Enterprise           OK; bingo; keep it
1|James T.|Kirk|1|2|USS Enterprise         Doesn't match; throw away
1|James T.|Kirk|1|3|Millennium Falcon      Oops ...
<<<<<<
4|Han|Solo|3|2|USS Enterprise               Not really ...
4|Han|Solo|3|3|Millennium Falcon           OK; bingo; keep it
4|Han|Solo|3|4|Death Star                  Sigh
<<<<<<                                     Hours later
6|Ma|Reynolds|5|5|Serenity                 Fine, keep this (Whew.)
```

In our toy example this isn't really a problem, but if you consider that the IRS might want to match tax payers to their employers, the dimensions are somewhat different.

This is why SQL lets you specify in advance which combinations of tuples you find interesting, instead of creating *all* possible combinations and then throwing out the uninteresting ones. This looks like

```
sqlite> SELECT *
...> FROM person JOIN ship ON person.ship_id=ship.id;
Result: see above
```

query optimisation 

Whether this is a real problem in practice also depends on your database system. A large part of the development effort for a database system goes into “query optimisation”, which is the part that decides exactly how to evaluate SELECT commands. Clever database systems can figure out that the first example and the JOIN example do essentially the same thing, and handle both in the same (efficient) manner. SQLite, for example, generates the same byte code for both queries. On the other hand, both these queries are still very simple, and in real life you will have to deal with more complex queries that may tax a query optimiser to a point where it no longer notices obvious simplifications. We recommend you use the JOIN form just to be safe.

If you want to know which commander appeared in which film, you will have to consult the personfilm table:

```
sqlite> SELECT firstname, name, title
...> FROM person p JOIN personfilm pf ON p.id=pf.person_id
...> JOIN film f ON pf.film_id=f.id;
James T.|Kirk|Star Trek
James T.|Kirk|Star Trek: Generations
Willard|Decker|Star Trek
Jean-Luc|Picard|Star Trek: Generations
Han|Solo|Star Wars 4
Han|Solo|Star Wars 5
Wilhuff|Tarkin|Star Wars 4
Mal|Reynolds|Serenity
```

So even relations between three (and more) tables are not a problem—you simply need to keep your eyes peeled!

Here are some more examples for relational queries. First the list of all commanders who appeared in films since 1980:

```
sqlite> SELECT year, title, firstname || ' ' || name
...> FROM person p JOIN personfilm pf ON p.id=pf.person_id
...> JOIN film f ON pf.film_id=m.id
...> WHERE year >= 1980 ORDER BY year ASC;
1980|Star Wars 5|Han Solo
1994|Star Trek: Generations|James T. Kirk
1994|Star Trek: Generations|Jean-Luc Picard
2005|Serenity|Mal Reynolds
```

Here is a list of films featuring two or more commanders:

```
sqlite> SELECT title
...> FROM film f JOIN personfilm pf ON f.id=pf.film_id
...> GROUP BY film_id HAVING COUNT(*) > 1;
Star Trek
Star Trek: Generations
Star Wars 4
```

The “GROUP BY” clause causes tuples from personfilm that refer to the same film to be processed together. HAVING, (which we didn’t cover before) is similar to WHERE, but it is applied after grouping and allows the use of aggregate functions (which WHERE doesn’t); hence the COUNT(*) in HAVING clause counts the tuples in each group.

Exercises



8.9 [!] What is the output of the SQL command

```
SELECT * FROM person, ship
```

when it is applied to our sample database?

8.5 Practical Examples

Now that you have looked into the basics of SQL, you may well wonder what all of this buys you in practice (unless you are working with databases already, in which case all of this chapter is probably old hat to you). In this section we present a few ideas of what to do with an SQL database system like SQL in “real life”.

Firefox As of version 3, Firefox (or, for Debian GNU/Linux users, “Iceweasel”) uses SQLite to manage an increasing number of its internal files. You can snoop around some of them and learn interesting things; anything in the `~/.mozilla/firefox/*.Default-User` directory (where the asterisk represents a code to make the name unique) with an extension of `.sqlite` is potentially fair game.

The `formhistory.sqlite` file, for example, contains the default values Firefox inserts into web forms. The database schema is rather obvious:

```
CREATE TABLE moz_formhistory (
  id INTEGER PRIMARY KEY,
  fieldname LONGVARCHAR,
  value LONGVARCHAR
);
```

So you can use

```
sqlite> SELECT value FROM moz_formhistory WHERE fieldname='address';
```

to find out what Firefox will propose to you if an input field in a web form has the (internal HTML) name `address`. Likewise, you have the possibility to systematically get rid of any default values that bug you (which Firefox itself doesn’t offer)—a suitable `DELETE FROM` creates *faits accomplis*.



If you want to be on the safe side, do this when Firefox isn’t running—but in principle it should work even if it is.

Also as of version 3, Firefox maintains a file named `places.sqlite`, which contains interesting things like your bookmarks (in `moz_bookmarks`), the sites you visited (in `moz_historyvisits` and `moz_places`), and much else.

Amarok Are you using the KDE music player, Amarok? If so, you can easily figure out your personal “tops of the pops” using SQL:

```
$ sqlite3 ~/.kde/share/apps/amarok/collection.db \  
> 'SELECT url, playcounter FROM statistics ORDER BY \  
> playcounter DESC LIMIT 10;'
```

The “LIMIT 10” clause at the end of the query limits the number of resulting tuples to at most 10. If you want to list the artist and title instead of the URL of the song file, the query is only a bit more complex;

```
$ sqlite3 ~/.kde/share/apps/amarok/collection.db \  
> 'SELECT title, playcounter FROM statistics s \  
> JOIN tags t ON s.url=t.url \  
> ORDER BY playcounter DESC LIMIT 10;'
```



Incidentally, a more convenient way to express the JOIN in the above query would be to use JOIN tags USING(url), since in both tables the url column is used to make the connection.

Do look at the Amarok schema using .schema. You will surely be able to think of other interesting applications.

Poor Person’s Diary If you are one of those people who tend to forget Auntie Mildred’s birthday, you should attempt, for the sake of the family peace, not to let this happen too often. It would be nice to be made aware of the upcoming family events and anniversaries—ideally with a little advance warning so you can still take care of gifts, cards, and so on.

Since you are familiar with the Unix shell, writing a small diary application to help you keep track should be a piece of cake. Of course we will not be able to compete with Evolution nor KOrganizer nor Google Calendar, but shall set our sights rather lower than that (but not too low).

The first thing we have to do is design the database schema. We would like to store different kinds of events (birthdays, anniversaries, etc) as well as, for each event, not just the category but also the date and an explanation (so we know what all of this is about). Here is a proposal for the category:

```
CREATE TABLE type (
  id INTEGER PRIMARY KEY,
  abbr VARCHAR(1),
  name VARCHAR(100)
);
```

abbreviation
full text

And here are the events themselves:

```
CREATE TABLE event (
  id INTEGER PRIMARY KEY,
  type_id INTEGER,
  year INTEGER,
  date VARCHAR(5),
  description VARCHAR(100)
);
```

foreign key

We store the date of each event separately; the year (year column) and the month and day in MM-DD format (date column), for reasons which will hopefully become clear soon. Entries are added to the diary as follows:

```
INSERT INTO type VALUES (1, 'B', 'Birthday');
INSERT INTO type VALUES (2, 'W', 'Wedding Anniversary');
INSERT INTO type VALUES (3, 'A', 'Anniversary');
```

```
INSERT INTO event VALUES (1, 1, 1926, '01-17', 'Auntie Mildred');
INSERT INTO event VALUES (2, 1, 1934, '01-21', 'Uncle Jack');
INSERT INTO event VALUES (3, 2, 2002, '02-05', 'Susie and Martin');
```

Now we can answer the pressing question: What will happen next week? In other words: If we “teleport” the dates from event into the current year, which of them fall into the range from “today” to “today plus seven days”? (We assume that one week is adequate to buy a card or a present. The stationery shop around the corner will surely have something in stock, and the likes of Amazon will, in a pinch, deliver stuff fairly quickly.)

Our task is simplified considerably by the date functions of SQLite, which we haven’t looked at before. The DATE() function, for example, expects as its first argument a date in the common “international” format (meaning first the year, then the month, and then the day, separated by hyphens—for instance, 2009-01-14) or the string now (with the obvious meaning). Any subsequent arguments “modify” the date given as the first argument. So you can obtain the point of time corresponding to “today plus seven days” simply using

```
sqlite> SELECT DATE('now', '+7 days');
2009-01-20          Today, incidentally, is 13 January 2009
```

This pretty much settles it: We can find all “current” dates using a query like

```
sqlite> SELECT DATE('2009-' || date) AS d, description, year
...> FROM event
...> WHERE d >= DATE('now') AND d <= DATE('now', '+7 days')
2009-01-17|Auntie Mildred|1926
```



Instead of

```
d >= DATE('now') AND d <= DATE('now', '+7 days')
```

you could also write

```
d BETWEEN DATE('now') AND DATE('now', '+7 days')
```

You can now wrap the query up nicely in a shell script, which will figure out the current year (somewhat cumbersome in SQL) and formats the output nicely (*quite* cumbersome in SQL). The result might look like figure 8.4, and together with a database file in ~/.cal.db might produce output like following form:

```
$ calendar-upcoming 60          upcoming two months
2009-01-17: Auntie Mildred (83th birthday)
2009-01-21: Uncle Jack (75th birthday)
2009-02-05: Susie and Martin (7th wedding anniversary)
2009-02-06: ROM-TOS anniversary (23th anniversary)
2009-02-17: Rabbit Keeping Club (124th anniversary)
```

There are many possible extensions. Just have a look at the exercises.


```
#!/bin/bash
# calendar-upcoming [limit]

caldb=$HOME/.cal.db
year=$(date +%Y)
limit=${1:-14}


sqlite3 $caldb \
  "SELECT DATE('$year-' || date) AS d, name, description, year
  FROM event JOIN type ON event.type_id=type.id
  WHERE d >= DATE('now') AND d <= DATE('now', '+$limit days')
  ORDER BY d ASC;" \
| awk -F'|' '{ print $1 ": " $3 " (" year-$4 "th " $2 ")" }' year=$year
```


Figure 8.4: The calendar-upcoming Script


Exercises

 **8.10** [3] Add a “holiday” type to the “poor man’s diary”. When a holiday is listed, no “age” should be added to the output:

```
2009-02-17: Rabbit Keeping Club (124th anniversary)
2009-02-23: Carnival Monday (holiday)    in parts of Germany
2009-03-01: Cousin Fred (29th birthday)
```

 **8.11** [3] Write a shell script named calendar-holidays which is passed the date of Easter in the usual ISO format (e.g.: 2009-04-12) and adds the holidays of the given year to the calendar database.

 **8.12** [4] (Previous exercise continued—this is a more elaborate project.) Write a program that computes Easter of the given year. Information about computing the date of Easter can be found here: <http://en.wikipedia.org/wiki/Computus>. (*Hint*: use awk.) Rewrite calendar-holidays such that it uses your new program—it will only require the year number rather than the date of Easter afterwards.

 **8.13** [2] Make the program mail you a list of the forthcoming events in each morning. (You will probably need some information from Chapter 9 to do this exercise, so feel free to come back to it at a later time.)

Summary

- SQL (“Structured Query Language”) is a standard language for defining, querying, and manipulating relational databases.
- Normalisation is an important process for maintaining the consistency of relational databases.
- There are various SQL database products for Linux systems.
- A collection of table definitions is called a “database schema”.
- SQL does not only support defining database schemas, but also inserting, modifying, querying, and deleting data (tuples).
- Aggregate functions allow you to compute values like the sum or average of all tuples of a column.
- Relations allow you to process data from multiple tables.
- Many Linux-based applications programs use SQLite for storing data.

