3

DATABASES

There is a way between voice and presence
Where information flows.

—Rumi (2004, p 32)

Structured Query Language (*SQL*[1]) is a specialized language that deals only with the flow of information. Some things, like joining together multiple data sets, are a pain using traditional techniques of matrix manipulation, but are an easy *query* in a database language. Meanwhile, operations like matrix multiplication or inversion just can not be done via SQL queries. With both database tables and C-side matrices, your data analysis technique will be unstoppable.

As a broad rule, try to do data manipulation, like pulling subsets from the data or merging together multiple data tables, using SQL. Then, as a last step, pull the perfectly formatted data into an in-memory matrix and do the statistical analysis.

Because SQL is a specialized language that deals only with information flows, it is not nearly as complex as C. Here is some valid SQL: `select age, gender, year from survey`. That's almost proper English. It goes downhill from there in terms of properness, but at its worst, it is still not difficult to look at an SQL query and have some idea of what the rows and columns of the output table will look like.

[1] Some people pronounce SQL as *sequel* and some as *ess queue ell*. The official ISO/IEC standard has no comment on which is correct.

Like C, SQL is merely a language, and it is left to the programmers of the world to write code that can parse SQL and return data from SQL queries. Just as this book leans toward `gcc` to interpret C code, it recommends the SQLite library, by D Richard Hipp, to interpret code written in SQL. SQLite provides a library of functions that parse SQL queries and uses those instructions to read and write a specific format of file [see binary trees in Chapter 6]. Any program that uses the SQLite function library is reading and writing the same file format, so SQLite files can be traded among dozens of programs.

As with C utilities, the only problem is selecting which SQLite database viewer to use among the many options. The SQLite library comes with a command-line program, `sqlite3`, but there are many other alternatives that are more reminiscent of the table view in the standard stats package or spreadsheet; ask your search engine for *sqlite browser* or *sqlite GUI*. These programs will give you immediate feedback about any queries you input, and will let you verify that the tables you are creating via C code are as you had expected.

Why is SQLite *lite*? Because most SQL-oriented databases are designed to be used by multiple users, such as a firm's customers and employees. With multiple users come issues of simultaneous access and security, that add complications on top of the basic process of querying data. SQLite is designed to be used by one user at a time, which is exactly right for the typical data analysis project. If you hope to use another database system, you will need to learn the (typically vendor-specific) commands for locking and permissions.

This chapter will primarily consist of an overview of SQL, with which you can follow along using any of the above tools. Section 3.5 will describe the Apophenia library functions that facilitate using an SQL database such as SQLite or mySQL from within a C program.

$\mathbb{Q}_{3.1}$

Check that both the SQLite executable and development libraries are correctly installed. In the online code supplement, you will find an SQLite-formatted database named `data-wb.db` listing the 2005 GDP and population for the countries of the world. Verify that you can open the database using one of the above tools (e.g., `sqlite3 data-wb.db` from the command prompt), and that you can execute and view the results of the query `select * from pop;`.

Once you have a working query interpreter, you can follow along with the discussion in this chapter. For your cutting and pasting convenience, most of the queries in this chapter are also in the `queries` file in the online code supplement.

Data format A database holds one or more tables. Each column in a table represents a distinct variable. For example, a health survey would include columns such as subject's age, weight, and height. Expect the units to be different from column to column.

Each row in a table typically represents one observation. For example, in a survey, each row would be data about a single person. There is no mechanism in SQL for naming a row, although it is common enough to have a plain column named `row_name`, or another identifier such as `social_security_no` that serves this purpose.

The asymmetry between columns and rows will be very evident in the syntax for SQL below. You will select columns using the column name, and there is no real mechanism for selecting an arbitrary subset of columns; you will select rows by their characteristics, and there is no real mechanism to select rows by name.[2]

Your C-side matrices will generally be expected to have a similar format; see page 147 for further notes.

Most of the world's data sets are already in this format. If your data set is not, your best bet is to convert it rather than fighting SQL's design; see the notes on crosstabs, page 101, for tips on converting from the most common alternative data format.

3.1 BASIC QUERIES

SQL's greatest strength is selecting subsets of a data set. If you need all of the data for those countries in the World Bank data set (`data-wb.db`) with populations under 50 million, you can ask for it thusly:

```
select *
from pop
where population <= 50;
```

You can read this like English (once you know that `*` means 'all columns'): it will find all of the rows in a table named `pop` where `population` in that row is less than or equal to 50, and return all the columns for those rows.

[2] If there is a `row_name` variable, then you could select rows `where row_name = 'Joe'`, but that is simply selecting rows with the characteristic of having a `row_name` variable whose value is `'Joe'`. That is, column names are *bona fide* names; row names are just data.

Generally, the `select` statement gives a list of columns that the output table will have; the `from` clause declares where the source data comes from; and the `where` clause lists restrictions on the rows to be output. And that's it. Every query you run will have these three parts in this order: column specification, data source, row specification.[3] This simple means of specifying rows, columns, and source data allows for a huge range of possibilities.

Commas and semicolons

In SQL, semicolons are *terminators* for a given command. You can send two SQL commands at once, each ending with a semicolon. Many SQLite-based programs will forgive you for omitting the final semicolon.

Commas are *separators*, meaning that the last element in a comma-separated list must not have a comma after it. For example, if you write a query like `select country, pop, from population` then you will get an error like "syntax error near `from`" which is referring to the comma just before `from` that is not separating two columns.

Select The `select` clause will specify the columns of the table that will be output. The easiest list is `*`, which means 'all the columns'. Other options:

- Explicitly list the columns:
  `select country, population`
- Explicitly mention the table(s) from which you are pulling data:
  `select pop.population, gdp.country`
  This is unnecessary now, but will become essential when dealing with multiple tables below.
- Rename the output columns:
  `select pop.country as country, gdp as gdp_in_millions_usd`
  If you do not alias `pop.country as country`, then you will need to use the name `pop\.country` in future queries, which is a bit annoying.
- Generate your own new columns. For example, to convert GDP in dollars to GDP in British pounds using the conversion rate as of this writing:
  `select country, gdp*0.506 as gdp_in_GBP`
  The `as gdp_in_GBP` subclause is again more-or-less essential if you hope to refer to this column in the future.

From The `from` clause specifies the tables from which you will be pulling data. The simplest case is a single table: `from data_tab`, but you can specify as many tables as necessary: `from data_tab1, data_tab2`.

You can alias the tables, for easier reference. The clause `from data_tab1 d1,`

[3] You may have no row restrictions, in which case your query will just have the first two parts and a null third part.

`data_tab2 d2` gives short names to both tables, which can be used for lines like `select d1.age, d2.height`.

Another option is to take data from subqueries; see below.

Borrowing C's annoyances

SQL accepts C-style block comments of the form `/* ... */`. It has the same trouble with nested block comments as C (see p 25). With one-line comments, everything after two dashes, `--`, is ignored, comparable to the two slashes, `//`, in C. [mySQL users will need two dashes and a space: `-- `.]

Also following C's lead, dividing two integers produces an integer, not the real number we humans expect. Thus, rather than calculating, say, `count1/count2`, cast one of the columns to a real number by adding `0.0`: `(count1+0.0)/count2` will return the real number it should. The add-zero trick also works to turn the string `"1990"` into the number `1990`. [SQL has a `cast` keyword, but it is much easier to just use the trick of adding `0.0`.]

Aliasing is generally optional but convenient, but one case where it is necessary arises when you are joining a table to itself. For now, simply note the syntax: `from data t1, data t2` will let you refer to the `data` table as if it were two entirely independent tables.

Notice, by the way, that when we aliased something in the `select` section, the form was `select long_col_description as lcd`, while in the `from` section there is no *as*: `from long_file_name lfn`.[4]

Where The `where` clause is your chance to pick out only those rows that interest you. With no `where` clause, the query will return one line for every line in your original table (and the columns returned will match those you specified in the `select` clause). For example, try `select 1 from gdp` using the `data-wb.db` database.

You can use the Boolean operators you know and love as usual: `where ((d1.age > 13) or (d2.height >= 175)) and (d1.weight = 70)`. SQL does not really do assignments to variables, so the clause `(d1.weight = 70)` is a test for equality, not an assignment. SQLite is easygoing, and will also accept the C-format `(d1.weight == 70)`; other SQL parsers (like mySQL) are less forgiving and consider the double-equals to be an error.

- You can select based on text the same way you select on a number, such as `where country = 'United States'`. Any string that is not an SQL keyword or the name of a table or column must be in 'single-tick' quotation marks.[5]

[4] The `as` is actually optional in the `select` clause, but it improves readability.

[5] Again, SQLite is forgiving, and will also accept C-style "double-tick" quotation marks. However, it is beneficial that SQL uses single-ticks while C uses double-ticks, because `snprintf(q, 100, "select * where country = 'Qatar'")` requires no unsightly backslashes, while double-tick quotation marks do: `snprintf(q, 100, "select * where country = \"Qatar\"")`.

- Case matters: `'United States' != 'united states'`. However, there is an out should you need to be case-insensitive: the `like` keyword. The clause `where country like 'united states'` will match the fully-capitalized country name as well as the lower case version. The `like` keyword will even accept two wild cards: _ will match any single character, and % will match any set of characters. Both `country like 'unit%ates'` and `country like 'united_states'` will match `'United States'`.
- The `where` clause refers to the root data, not the output, meaning that you can readily refer to columns that you do not mention in the `select` clause.

$\mathbb{Q}_{3.2}$ Use a `where` clause and the `population` table to find the current population of your home country. Once you know this amount, select all of the countries that are more populous than your country.

Generalizing from equality and inequalities, you may want a group of elements or a range. For this, there are the `in` and `between` keywords. Say that we want only the United States and China in our output. Then we would ask only for columns where the country name is in that short list:

```
select *
from gdp
where country in ("United States", "China")
```

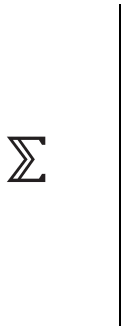
The `in` keyword typically makes sense for text data; for numeric data you probably want a range. Here are the countries with GDP between $10 and $20 billion:

```
select *
from gdp
where gdp between 10000 and 20000
```

$\mathbb{Q}_{3.3}$ Write a query using `<=` and `>=` to replicate the above query that used `between`.

$\sum$

- A query consists of three parts: the columns to be output, the data source, and the rows to be output.
- The columns are specified in the `select` statement. You can pull all the columns from the data using `select *`, or you can specify individual columns like `select a, b, (a+0.0)/b as ratio`. ⋙



- The data source is in the `from` clause, which is typically a list of tables.
- The row specification, generally in the `where` clause, is a list of conditions that all rows must meet. It can be missing (and so all possible rows are returned) or it can include a series of conditions, like `where (a = b) and (b <= c)`.

3.2 ※ DOING MORE WITH QUERIES

Beyond the basic `select - from - where` format, a `select` query can include several auxiliary clauses to refine the output further. Here is the complete format of a `select` query, which this section will explore clause by clause.

select [**distinct**] columns
from tables
where conditions
group by columns
having group_conditions
order by columns
limit n **offset** n

PRUNING ROWS WITH `distinct`

The `data-metro.db` file includes a listing of all stations and the color of the subway line(s) on which the station lies. The query `select line from lines` produces massive redundancy, because there are a few dozen stations on every line, so each color appears a few dozen times in the table.

The `distinct` keyword will tell the SQL engine that if several rows would be exact duplicates, to return only one copy of that row. In this case, try

select distinct line
from lines

The `distinct` word prunes the rows, but is placed in the `select` portion of the program. This reads more like English, but it breaks the story above that the `select` statement specifies the columns and the `where` statement specifies the rows.

AGGREGATION Here is how to get the number of rows in the `gdp` table of `data-wb.db`:

```
select count(*) as row_ct
from gdp;
```

This produces a table with one column and one row, listing the total number of rows in the `data` table.

Q3.4 How many rows does `select * from pop, gdp` produce? The explanation for the answer will appear in the section on *joins*, below.

You probably want more refinement than that; if you would like to know how much data you have in each region, then use the `group by` clause to say so:

```
select class, count(*) as countries_per_class
from classes
group by class;
```

After `count`, the two most common aggregation commands are `sum()` and `avg()`. These take an existing row as an argument. For example, the `data-tattoo.db` database has a single table representing a telephone survey regarding tattoos. To get the average number of tattoos per person broken down by race, you could use this query:

```
select race, avg(tattoos.'ct tattoos ever had')
from tattoos
group by race;
```

Feel free to specify multiple `group by` clauses. For example, you could modify the above query to sort by race and age by changing `group by race` to `group by race, tattoos.'year of birth'`. When you want to analyze the output, you will be very interested in the `apop_db_to_crosstab` function; see page 101.

Q3.5 In the `precip` table of the `data-climate.db` database, the `yearmonth` column encodes dates in forms like `199608` to mean August, 1996. Fortunately, the SQL-standard `round()` function can be used to produce a plain year: `round(199608./100.) == 1996.0`. Use `round`, `group by`, and `avg` to find the average precipitation (`pcp`) in each year.

You can use `count` with the `distinct` keyword to find out how many of each row you have in a table. This is useful for producing weights for each observation type,

Function	Standard SQL	mySQL	SQLite via Apophenia
abs, avg, count, max, min, round,[a] sum	○	○	○
acos, asin, atan, cos, exp, ln, log10, pow, rand, sin, sqrt, stddev_s, tan, variance_p, std_p, stddev_pop_p, stddev_samp_s, var_samp_s, var_pop_p		○	○
ran, var_s, skew_s, kurtosis_s, kurt_s			○

[a] Round is not part of the SQL standard, which instead provides `floor` and `ceil`.

Table 3.1 Standard SQL offers very few mathematical functions, so different systems offer different extensions. The p and s subscripts indicate functions for populations or for samples (see box on page 222.

as in this query to produce a tabulation of respondents to the tattoo survey by race and birth year:

```
select distinct race, tattoos.'year of birth' as birthyear, count(*) as weight
from tattoos
group by race, birthyear
```

With a `group by` command, you have two levels of elements, items and groups, and you may want subsets of each. As above, you can get a subset of the items with a `where` clause. Similarly, you can exclude some groups from your query using the `having` keyword. For example, the above query produced a lot of low-weighted groups. What groups have a `count(*) > 4`? We can't answer this using `where weight > 4`, because there is no `weight` column in the data table, only in the post-aggregation table. This is where the `having` keyword comes in:

```
select distinct race, tattoos.'year of birth' as birthyear, count(*) as weight
from tattoos
group by race, birthyear
having weight > 4
```

※ *SQL extensions* That's all the aggregators you get in standard SQL. So implementers of the SQL standard typically add additional functions beyond the standard; see Table 3.1 for a list, including both aggregation functions

like `var` and $\mathbb{R} \to \mathbb{R}$ functions like `log`. The table focuses on numeric functions, and the standard and mySQL both include several functions for manipulation of text, dates, and other sundry types of data; see the online references for details.

Bear portability in mind when using these functions, and be careful to stick to the SQL standard if you ever hope to use your queries in another context. If you want to stay standard, call your data into a C-side vector or matrix and use `apop_vector_log`, `apop_vector_exp`, `apop_vector_skew`, `apop_vector_var`, ..., to get the desired statistics on the matrix side.

SORTING To order the output table, add an `order by` clause. For example, to view the list of country populations in alphabetical order, use

```
select *
from pop
order by country
```

- You may have multiple elements in the clause, such as `order by country, pop`. If there are ties in the first variable, they are broken by the second.
- The keyword `desc`, short for *descending*, will reverse the order of the variable's sorting. Sample usage: `order by country desc, pop`.

GETTING LESS Especially when interactively interrogating a database, you may not want to see the whole of the table you have constructed with a `select` clause. The output may be a million lines long, but twenty should be enough to give you the gist of it, so use a `limit` clause. For example, the following query will return only the first twenty rows of the `pop` table:

```
select *
from pop
limit 20
```

You may want later rows, and so you can add the `offset` keyword. For example,

```
select *
from pop
limit 5 offset 3
```

will return the first five rows, after discarding the first three rows. Thus, you will see rows 4–8. Beyond making interactive querying easier, `limit - offset`

clauses can also be used to break tables that are somehow giving you problems into more manageable pieces, probably via a C-side `for` loop.

- You get one `limit/offset` per query, which must be the last thing in the query.
- If you are using `union` and family to combine `select` statements (see below), your `limit` clause should be at the end of all of them, and applies only to the aggregate table.

※ *Random subsets* The `limit` clause gives you a sequential subset of your data, which may not be representative. If this is a problem, you can take a random draw of some subset of your data. Ideally, you could provide a query like `select * from data where rand() < 0.14` to draw 14% of your data.

SQLite-via-Apophenia and mySQL provide a `rand` function that works exactly as above.[6] For every call to the function (and thus, for every row), it draws a uniform random number between zero and one.[7]

CREATING TABLES There are two ways to create a table. One is via a `create` statement and then an `insert` statement for every single row of data. The `create` statement requires a list of column names;[8] the `insert` statement requires a list of one data element for each column.

```
begin;
create table newtab(name, age);
insert into newtab values("Joe", 12);
insert into newtab values("Jill", 14);
insert into newtab values("Bob", 14);
commit;
```

The `begin-commit` wrapper, by the way, means that everything will happen in memory until the final commit. The program may run faster, but if the program

[6] Standard SQL's `random` function is absolutely painful. SQLite's version currently produces a number between ±9,223,372,036,854,775,807, which the reader will recognize as $\pm(2^{63}-1)$. So we need to pull a random number, divide by $2^{63}-1$, shift it to the familiar $[0, 1]$ range, and then compare it to a limit. Standard SQL does not even provide exponentiation, so doing this requires the bit-shifting operator which I had promised you would never need; read `1<<x` as 2^x. That said, `select * from data where (random()/(-(1<<63)-1.0)+1)/2 < 0.14` will pull approximately 14% of the data set.

[7] After you read Section 11.1, you will wonder about the stream of random numbers produced in the database. There is one stream for the database, which Apophenia maintains internally. To initialize it with a seed of seven, use `apop_db_rng_init(7)`. If you do not call this function, the database RNG auto-allocates at first use with seed zero.

[8] SQLite has the pleasant property that its columns are basically type-less. Other database engines insist on table declarations that look a little like C functions, e.g., `create table newtab(name varchar[30], age int)`; see your database engine documentation for details.

crashes in the middle, then you will have lost everything. The optimal speed/security trade-off is left as an exercise for the reader.

If you have hundreds or thousands of `inserts`, you are almost certainly better off putting the data in a text file and using either the C function `apop_text_to_db` or the command-line program with the same name. The form above is mostly useful in situations where you are creating the table in mid-program, as in the example on page 108.

The other method of creating a table is by saving the results of a query. Simply put `create table` *`newtab_name`* `as` at the head of the query you would like to save:

```
create table tourist_traps as
    select country
    from lonely_planet
    where (0.0+pp) > 600
```

$\mathbb{Q}_{3.6}$ The `riders` table of the `data-metro.db` database includes the average boardings in each station of the Washington Metro system, every year since its opening. Create a `riders_per_year` table with one column for the year and one column for total average boardings across the system for the given year.

DROPPING A TABLE The converse of table creation is table dropping:

```
drop table newtab;
```

See also `apop_table_exists` on the C-side (p 108), which can also delete tables if desired.

ROWID Sometimes, you need a unique identifier for each output row. This would be difficult to create from scratch, but SQLite always inserts such a row, named `rowid`. It is a simple integer counting from one up to the number of rows, and does not appear when you query `select * from table`. But if you query `select rowid, * from table`, then the hidden row numbers will appear in the output.[9]

[9] mySQL users will need to explicitly ask for such a column when creating the table. A statement like `create table` *`newtab`* `(`*`id_column`* `int auto_increment,` *`info1`* `char(30),` *`info2`* `double, ...)` will create the table with the typical columns that you will fill, plus an *`id_column`* that the system will fill. After `insert into newtab values ("Joe", 23); insert into newtab values ("Jane" 21.8);`, the table will have one row for Joe where *`id_column`*`==1` and one for Jane where *`id_column`*`==2`.

$\mathbb{Q}_{3.7}$ Using `order by` and `rowid`, find the rank of your home country's GDP among countries in the World Bank database.

METADATA What tables are in the database? What are their column names? Standard SQL provides no easy way to answer these questions, so every database engine has its own specific means. SQLite gives each database a table named `sqlite_master` that provides such information. It includes the type of object (either index or table, in the `type` column), the name (in the `name` column), and the query that generated the object (in the `sql` column). MySQL users, see page 106.

In practical terms, this table is primarily good for getting the lay of an unfamiliar database—a quick `select * from sqlite_master;` when you first open the database never hurts. If you are using the SQLite command line, there is a `.table` command that does exactly what this program does. Thus, the command `sqlite3` *`mydb.db`* `.table` just lists available tables, and the `.schema` command gives all of the information from `sqlite_master`.

MODIFYING TABLES SQL is primarily oriented toward the filtering style of program design: e.g., have one query to filter a data table to produce a new table with bad data removed, then have another query to filter the resulting table to produce an aggregate table, then select some elements from the aggregate table to produce a new table, et cetera.

But you will often want to modify a table in place, rather than sending it through a filter to produce a new table (especially if the table is several million entries long). SQL provides three operations that will modify a table in place.

`delete` Unlike `drop`, which acts on an entire table, `delete` acts on individual rows of a database. For example, to remove the columns with missing GDP data, you could use this query [—but before you destroy data in the sample databases, make a copy, e.g., via `create table gdp2 as select * from gdp`]:

```
delete from gdp
where gdp='..'
```

`insert` The obvious complement to deleting lines is inserting them. You already saw `insert` used above in the context of creating a table and then inserting elements item-by-item. You can also insert via a query, via the form `insert into` *`existing_table`* `select * from ...`.

`update` The `update` query will replace the data in a column with new data. For example, the World Bank refrained from estimating Iraq's 2006 population, but the US Central Intelligence Agency's *World Factbook* for 2006 estimates it at 26,783,383. Here is how to change Iraq's population (in the `pop` table) from `..` to `26783`:

```
update pop
set population=26783
where country='Iraq'
```

- You can limit your queries to fewer rows using a `limit` clause, which gives you a sequential snippet, or via random draws.
- The SQL standard includes a few simple aggregation commands: `avg()`, `sum()`, and `count()`, and most SQL implementations provide a few more nonstandard aggregators for queries called using its functions.
- When aggregating, you can add a `group by` clause to indicate how the aggregation should be grouped.
- Sort your output using an `order by` clause.
- You can create tables using the `create` and `insert` commands, but you are probably better off just reading the table from a text file. Use `drop` to delete a table.
- SQLite gives every row a `rowid`, though it is hidden unless you ask for it explicitly.

3.3 JOINS AND SUBQUERIES

So far, we have been cutting one table down, either by selecting a subset of rows or by grouping rows. SQL's other great strength is in building up tables by joining together data from disparate sources. The joining process is not based on a `join` keyword, but simply specifying multiple data sources in the `from` section of your query and describing how they mesh together in the `where` section.

If you specify two tables in your `from` line, then, lacking any restrictions, the database will return one joined line for every pair of lines. Let table 1 have one column with data $\begin{bmatrix}1\\2\\3\end{bmatrix}$ and table 2 have one column with data $\begin{bmatrix}a\\b\\c\end{bmatrix}$; then `select`

`* from table1, table2` will produce an output table with every combination, $3 \times 3 = 9$ rows:

```
1 a
1 b
1 c
2 a
2 b
2 c
3 a
3 b
3 c.
```

Such a product quickly gets overwhelming: in the exercise on page 81, you saw how joining the 208 countries in the World Bank data's `pop` table with the same 208 countries in the `gdp` table produces a few hundred pages of rows.

Thus, the `where` clause becomes essential. Its most typical use for a join arises when one column in each table represents identical information. Out of the 43,264 rows from the above join, including those that matched Qatar with Ghana and Cameroon with Zimbabwe, we are interested only in those that match Qatar with Qatar, Cameroon with Cameroon, and so on. That is, we want only those rows `where pop.country = gdp.country`, and so the query makes sense only when that restriction is added in:

```
select pop.country, pop.population, gdp.GDP
    from pop, gdp
    where pop.country = gdp.country
```

You can see that using the table-dot-column format for the column names is now essential. In the `select` clause specifying the output columns, you can use either `pop.country` or `gdp.country`, since the two will be by definition identical, or if you are unconcerned with the country names and just want the numeric data you can omit names entirely.

$\mathbb{Q}_{3.8}$ Add a calculation to the `select` portion of the above query to find the GDP per capita of each country. Be sure to give the calculated column a name, like `gdp_per_cap` so you can `order by gdp_per_cap`.

$\mathbb{Q}_{3.9}$ The World Bank data includes a classification for each country. Countries receiving World Bank assistance (what the WB calls *client countries*) are classed by region (e.g., Middle East and North Africa), while other countries are binned into a generic class like "Lower-middle-income economies." Find the total GDP per capita for each World Bank grouping. Here, you will join using the country columns in the `gdp` and `classes` table, and by the country columns in the `pop` and `classes` table. Add up total GDP in the region, and divide by total population in the region.

Example: a time lag The form above, where two columns match, is by far the most common type of join, but there are other creative uses of joins. For example, it is common in time series analysis to include the value of a variable at time $t-1$ as data that influenced the value at time t.

The `data-climate.db` database includes a table of the deviation from the century-long norm for aggregate worldwide temperatures (see Smith & Reynolds (2005) for methods, caveats, and discussion). A quick `select * from temp` will show that there is an upward trend in the data: the first few years are all below zero; the last few years hover around 0.5.[10]

What does the month-to-month change look like? The first step is dealing with the fact that there are separate `year` and `month` columns. One solution would be to deal only with `year + month/12.`, which moves through time in smooth increments of $\frac{1}{12}$. This creates its own problem, because comparing floating-point values is not reliable: `1900 + 1./12. - 1./12.` could wind up as something like `1900.00001`, and a test whether this value exactly equals `1900` will fail. As a variant that solves this problem, instead of dividing months by 12, multiply years by 12, so that we are comparing only integers:

```
select R.year+R.month/12., R.temp − L.temp
from temp L, temp R
where R.year∗12 +R.month = L.year∗12 +L.month +1;
```

The salient feature of this data set is that not much happens. The long-term shift is the result of a large number of very small month-to-month changes.

[10] Chapter 5 will cover graphing, but for now, try `apop_plot_query data-climate.db "select temp from temp"` from your command line to get a visual indication of the trend.

$\mathbb{Q}_{3.10}$

Perhaps we would see a larger change via a larger time span. Calculate the year-to-year differences.

- Create an `annualized` table with two columns: the year and average `temp` over all months for the year.
- Join that table with itself lagged by one year. You won't have to worry about unreliable float comparisons, but recall that if SQLite thinks `year` is a string, then it will treat `year+0.0` as a number.

Having looked at year-long differences, try decades.

- Create a `decades` table with the average for each decade. (*Hint*: `group by round(year/10)`.)
- Join the table with itself lagged by ten years. Are the differences beginning to show a pattern?

Given that the data is sorted, we could also have done the matching of rows using the `rowid`:

```
select L.temp − R.temp
from temp L, temp R
where R.rowid+0.0=L.rowid−1;
```

SPEEDING IT UP Now that you have seen how to join tables, we now cover how to avoid joining tables. If two tables have a million elements each, then joining them using a clause like `where a=b` requires $1e6 \times 1e6 = 1e12$ (a trillion) comparisons. This is impossibly slow, so there are a number of tricks to avoid making all those $1e12$ comparisons.[11]

Indices You can ask the SQL engine to create an index for a table that you intend to use in a join later. The commands:

```
create index pop_index on population(country)
create index gdp_index on gdp(country)
```

[11] Say that you mean to join a million subjects via ID number, via `select t1.*, t2.* from t1, t2 where t1.id = t2.id`, but you forget to include the `where` clause. Then you just asked the system to create a trillion-entry table, which will take from several hours to weeks. Thus, the first step in speeding up an inordinately slow query is not to try the tricks in this section, but to make sure that you actually wrote the query you had intended to write.

would index the `pop` and `gdp` tables on the `country` column. The name of the index, such as `pop_index`, is basically irrelevant and can be any gibberish that sounds nice to you. Once you have created this index, a join using any of the indexed columns goes *much* faster, because the system no longer has to do `1e12` comparisons. Basically, it can look at the first value of *var* in the left table—say it is 17—and then check the right table's index for the list of elements whose value is 17. That is, instead of one million comparisons to join the first element, it only has to do one index lookup. The lookup and the process of building the tree took time as well, but these processes are on the order of millions of operations, not millions squared. The tree is internally structured as a binary tree; see Chapter 6 for discussion of b-trees.

There is standard SQL syntax for indexing multiple columns, e.g., `create index pop_index2 on pop(country, population)`, which goes by *lexicographic order*. This is just an index on the first item (`country`) with the second column (`population`) as a backup ordering; if you want to join by the second column, you should prepare by creating another index that puts that column in the first (or the only) position.

Subqueries Among SQL's nicest tricks is that it allows for the input tables to be queries themselves. For example: how large is the average World Bank grouping? Answering this question is a two-step process: get a `count(*)` for each category, and then get an average of that. You could run a query to produce a table of counts, save the table, and then run a query on that table to find the averages.

```
create table temptab as
    select count(*) as ct
      from classes
      group by class;
select avg(ct)
    from temptab
```

But rather than generating a temporary table, SQL allows you to simply insert the `select` statement directly into the query where it is used:

```
select avg(ct)
    from (select count(*) as ct
      from classes
      group by class)
```

The query inside the `from` clause will return a table, and even though that table has no name, it can be used as a data source like any other table. If the query output needs a name, you can alias the result as usual: `from (select ...) t1` will allow you to refer to the query's output as `t1` elsewhere in the query.

$\mathbb{Q}_{3.11}$ On page 79, you first found your home country's population, then the countries with populations greater than this. Use a subquery to do this in one query. (*Hint*: you can replace a number with a query that returns one element.)

Subsetting via a foreign table If you look at the World Bank data, you will see a large number of countries that are small islands of a few million people. Say that we are unconcerned with these countries, and want only the GDP of countries where `population > 270`.

$\mathbb{Q}_{3.12}$ Write a query to pull only the GDP of countries where the population is greater than 270 million, using the standard `where` *`leftcol=rightcol`* join syntax from the head of this section.

But the full join (as per the exercise) is not necessary: we are not particularly concerned with the population *per se*, but are just using it to eliminate rows. It would thus be logical to fit the query into the `where` clause, since that is the clause that is typically used to select a subset of the rows. Indeed, we can put a query directly into a `where ... in` clause:

```
select *
from gdp
where country in (select country from pop where population > 270)
```

The subquery will return a list of country names, and the main query can then use those as if you had directly typed them in.

This is typically much faster than a full join operation, because there was no need to make (left table row count) $\times$ (right table row count) comparisons.

The boost in efficiency implies some slight restrictions: because the `from` clause does not list the table used in the subquery, you can not refer to any of the subquery's columns in the output.

※ *Joining via a* `for` *loop* The time it takes to do an especially large join is not linear in the number of rows, primarily for real-world reasons of hardware and software engineering. If your computer can not store all the data points needed for a query in fast memory, it will need to do plenty of swapping back and forth between different physical locations in the computer. But your computer may be able to store a hundredth or a thousandth of the data set in fast memory, and so you can perhaps get a painfully slow query to run in finite

time by breaking it down into a series of shorter queries.

Here is an example from my own work (Baum *et al.*, 2008). We had gathered 550,000 genetic markers (SNPs) from a number of pools of subjects, and wanted the mean for each pool. Omitting a few details, the database included a `pools` table with the subject `id` and the `poolid` of its pool, with only about a hundred elements; and a table of individual `ids`, the `SNP` labels, and their values, which had tens of millions of values. Even after creating the appropriate indices, the straight join—

```
select pools.poolid as poolid, SNP, avg(val) as val, var(val) as var
from genes, pools
where genes.id=pools.id
group by pools.poolid, SNP
```

—was taking hours.

Our solution was to use a C-side `for` loop, plus subsetting via a foreign table, to avoid the join that was taking so long. There are three steps to the process: create a blank table to be filled, get a list of `poolids`, and then use `insert into ... select ...` to add each `poolid`'s data to the main table. The details of the functions will be discussed below, but these three steps should be evident in this code snippet.

```
apop_query("create table t (poolname, SNP, val, var);");
apop_data *names = apop_query_to_text("select distinct poolid from pools");
for (int i=0; i< names->textsize[0]; i++)
    apop_query("insert into t \n\
        select '%s', SNP, avg(val), var(val) \n\
        from genes \n\
        where id in (select id from pools where poolid = '%s') \n\
        group by SNP; \n\
            ", names[i][0], names[i][0]);
```

This allowed the full aggregation process to run in only a few minutes. The next week we bought better hardware.

As noted above, if there is no natural grouping like the pools in this example, a `for` loop using the `limit ...offset` form can also break a too-long table into smaller pieces.

STACKING TABLES You can think of joining two tables as setting one table to the right of another table. But now and then, you need to stack one on top of the other. There are four keywords to do this.

- `Union`: Sandwiching `union` between two complete queries, such as

```
select id, age, zip
from data_set_1
union
select id, age, zip
from data_set_2
```

  will produce the results of the first query stacked directly on top of the second query. Be careful that both tables have the same number of columns.
- `Union all`: If a row is duplicated in both tables, then the `union` operation throws out one copy of the duplicate lines, much like `select distinct` includes only one of the duplicates. Replacing `union` with `union all` will retain the duplicates.
- `Intersect`: As you can guess, putting `intersect` between two `select` statements returns a single copy of only those lines that appear in both tables.
- `Except`: This does subtraction, returning only elements from the first table that do not appear in the second. Notice the asymmetry: nothing in the second table will appear.

- ➤ You can put the output of a query into the `from` clause of a parent query.
- ➤ You can join tables by listing multiple tables in the `from` clause. When you do, you will need to specify a `where` clause, and possibly the `distinct` keyword, to prevent having an unreasonably long output table.
- ➤ If you intend to join elements, you can speed up the join immensely by creating an index first.
- ➤ If the join still takes too long, you can sidestep it via the `select ... where col in (select ...)` form, or via a C-side `for` loop.
- ➤ Tables can be stacked using `union`, `union all`, `intersect`, and `except`.

3.4 ON DATABASE DESIGN

Say that you are not reading in existing data, but are gathering your own, either from a simulation or data collected from the real world. Here are some considerations and suggestions for how you could design your database, summarizing the common wisdom about the best way to think about database tables.

The basic premise is that each type of object should have a single table, and each object should have a single row in that table.

Figure 3.2 shows a table of observations for a generic study involving several subjects and treatments, whose information was measured at several times. The simple one-table design is how the typical spreadsheet is designed. This version has one row per subject, so each row has two observations, and information about subjects, treatments, observations, and pools are mixed together.

Figure 3.3 shows a structure better suited for databases. For most statistical studies, the key object is the observation, and that gets its own table; we now see that there were twenty observations. The other objects in the study—subjects, pools, and treatments—all get their own tables as well. By giving each element of each table an ID number, each table can easily cross-reference others. This setup has many advantages.

Minimize redundancy This is rule number one in database design, and many a book and article has been written about how one goes about reducing data to the redundancy-minimized *normal form* (Codd, 1970). If a human had to enter all of the redundant data, this creates more chances for error, and the same opportunities for failure come up when the data needs to be modified when somebody notices that there were actually nine subjects in the pool from 6/2/02. In the single-table form, information about the pool was repeated for every member of the pool, while having a separate table for pools means that each pool's information is listed exactly once.

Ask non-observation questions There are reasons to ask questions based on treatments or pools, but a setup with only an observation-based table does not facilitate this. From the multiple tables, it is easy to ask questions that focus on data, treatments, or pools, via join operations on the observation, pool, subject, or treatment IDs.

Gelman & Hill (2007, p 239) point out that separating subjects and groups facilitates multilevel models, where each group has parameters for its own submodel estimated, and then those parameters are used to estimate an overall model. This sort of modeling will be covered in later chapters.

Use the power of row subsets Figure 3.2 includes multiple observations on one line, for the morning and evening measurements. But what if we went from two observations to hourly observations for 24 hours? Remember, there is no way to arbitrarily select a subset of columns, so columns

subjid	value_morn	value_eve	poolcount	pooldate	t_type	t_dosage
1	23.28	NaN	12	2/2/02	control	NaN
2	14.07	NaN	12	2/2/02	control	NaN
3	20.98	NaN	12	2/2/02	control	NaN
4	12.12	NaN	12	2/2/02	control	NaN
5	30.28	28.11	11	4/2/02	case	0.2
6	22.15	14.05	11	4/2/02	case	0.2
7	19.78	12.54	8	4/2/02	case	0.4
8	21.53	9.01	8	4/2/02	case	0.4
9	27.42	23.20	19	6/2/02	case	0.2
10	18.57	12.29	19	6/2/02	case	0.2

Figure 3.2 Spreadsheet style: one monolithic table, with much redundancy.

obsid	subjid	value	time
1	1	23.28	morn
2	2	14.07	morn
3	3	20.98	morn
4	4	12.12	morn
5	5	30.28	morn
6	6	22.15	morn
7	7	19.78	morn
8	8	21.53	morn
9	9	27.42	morn
10	10	18.57	morn
11	1	NaN	eve
12	2	NaN	eve
13	3	NaN	eve
14	4	NaN	eve
15	5	28.11	eve
16	6	14.05	eve
17	7	12.54	eve
18	8	9.01	eve
19	9	23.20	eve
20	10	12.29	eve

subjid	poolid	treatmentid
1	1	1
2	1	1
3	1	1
4	1	1
5	2	2
6	2	2
7	3	3
8	3	3
9	4	2
10	4	2

poolid	poolcount	pooldate
1	12	2/2/02
2	11	4/2/02
3	8	4/2/02
4	19	6/2/02

treatmentid	t_type	t_dosage
1	control	NaN
2	case	0.2
3	case	0.4
4	case	0.6

Figure 3.3 Database style: one table for each object type, one row for each object.

named `1AM`, `2AM`, ..., would be difficult to use. If we needed the mean of all morning observations, we'd need to do something like `select (12AM + 1AM + 2AM + 3AM + ...)/12`, but if the table in Figure 3.3 had an hour column, we could simply use:

```
select avg(value)
    from observations
    where time like '%am'
```

(or `where time < 12`, depending on the format we choose for the time).

If there is any chance that two observations will somehow be compared or aggregated, then they should probably be recorded in different rows of the same column. For 24 hours and ten subjects, the table would be 240 rows, which is not nearly as pleasing or human-digestible as a 10×24 spreadsheet. But you will rarely need to look at all the data at once, and can easily construct the crosstab if need be via `apop_db_to_crosstab`.

Even worse than having two data points of the same type in separate columns is having two data points of the same type in separate tables, such as a cases table and a controls table. Or, say that a political scientist wants to do a study of county-level data throughout the United States, including variables such as correlations between tax rates, votes by Senators, and educational outcomes. Because DC has no county subdivisions and its residents have no Congressional representation, the DC data does not fit the form of the data for the states and commonwealths of the United States. But the correct approach is nonetheless to put DC data in the same table as the counties of the fifty states, rather than creating a table for DC and a table for all other states—or still worse, a separate table for every state.

It is easy to `select * from alldata where senate_vote is not null` if DC's lack of representation will affect the analysis.[12]

- Databases are not spreadsheets. They are typically designed for many tables, which may have millions of rows if necessary.
- Each type of object (observations, treatments, groups) should have a single table, and each object should have a single row in that table.
- Bear in mind the tools you have when designing your table layouts. It is easy to join tables, find subsets of tables, and create spreadsheet-like crosstabs from data tables.

[12] By the way, `select * from alldata where population > (select population from alldata where state = 'DC')` won't work: it will return only 49 out of 50 states, because the population of DC (zero Senators, zero Representatives) is 572,000, while Wyoming (two Senators, one Representative) has a population of 494,000. [2000 census data]

3.5 FOLDING QUERIES INTO C CODE

This section covers the functions in the Apophenia library that will create and query a database. All of these functions are wrappers of functions in the SQLite or mySQL libraries that do the dirty work, but they are sufficiently complete that you should never need to use the functions in the SQLite/mySQL C libraries directly. The details of the main discussion will apply to SQLite; mySQL users, see page 106 for the list of differences.

IMPORTING The first command you will need is `apop_open_db`. If you give it the name of a file, like `apop_open_db("study.db")`, then the database will live on your hard drive. This is slower than memory, but will exist after you stop and restart the program, and so other programs will be able to use the file, you have more information for debugging, and you can re-run the program without re-reading in the data. Conversely, if you give a null argument—`apop_open_db(NULL)`—then the database is kept in memory, and will run faster but disappear when the program exits. Apophenia uses only one database at a time, but see the `apop_merge_dbs` and SQLite's `attach` functions below.

Command-line utilities

Apophenia includes a handful of command-line utilities for handling SQLite databases where there is no need to write a full-blown C program. `apop_text_to_db` reads a text file into a database table, `apop_merge_dbs` will send tables from one database to another, `apop_plot_query` will send query output directly to Gnuplot, and `apop_db_to_crosstab` will take a table from the SQLite database and produce a crosstab. All of these are simply wrappers for the corresponding Apophenia functions. For all of the utilities, you can use the `-h` parameter to get detailed instructions (e.g., `apop_plot_query -h`).

Unless your program is generating its own data, you will probably first be importing data from a text file. The `apop_text_to_db` function will do this for you, or you can try it on the command line (see box). The first line of the text file can be column names, and the remaining rows are the data. If your data file is not quite in the right format (and it rarely is), see Appendix B for some text massaging techniques.

When you are done with all of your queries, run `apop_close_db` to close the database. If you send the function a one—`apop_close_db(1)`—then SQLite will take a minute to clean up the database before exiting, leaving you with a smaller file on disk; sending in a zero doesn't bother with this step. Of course, if your database is in memory, it's all moot and you can forget to close the database without consequence.

The queries The simplest function is `apop_query`, which takes a single text argument: the query. This line runs the query and returns nothing, which is appropriate for `create` or `insert` queries:

```
int page_limit = 600;
apop_query(
        "create table tourist_traps as \
        select country \
        from lonely_planet \
        where (pp + 0.0) > %i ", page_limit);
```

- A string is easiest for you as a human to read if it is broken up over several lines; to do this, end every line with a backslash, until you reach the end of the string. The next example will use another alternative.
- As the example shows, all of Apophenia's query functions accept the `printf`-style arguments from page 26, so you can easily write queries based on C-side calculations.

  There are also a series of functions to query the database and put the result in a C-side variable. This function will run the given query and return the resulting table for your analysis:

```
int page_limit = 600;
apop_data *tourist_traps = apop_query_to_text(
        "select country "
        "from lonely_planet "
        "where (0.0+pp) > %i ", page_limit);
```

- C merges consecutive strings, so `"select country " "from"` will be merged into `"select country from"`. We can use this to split a string over several lines. But be careful to include whitespace: `"select country" "from"` merges into `"select countryfrom"`.

  After this snippet, `tourist_traps` is allocated, filled with data, and ready to use—unless the query returned no data, in which case it is `NULL`. It is worth checking for `NULL` output after any query that could return nothing. There are `apop_query_...` functions for all of the types you will meet in the next chapter, including `apop_query_to_matrix` to pull a query to a `gsl_matrix`, `apop_query_to_text` to pull a query into the text part of an `apop_data` set, `apop_query_to_data` to pull data into the matrix part, and `apop_query_to_vector` and `apop_query_to_float` to pull the first column or first number of the returned table into a `gsl_vector` or a `double`.

  For immediate feedback, you can use `apop_data_show` to dump your data to screen or `apop_data_print` to print to a file (or even back to the database). If you want a quick on-screen picture of a table, try

```
apop_data_show(apop_query_to_data("select * from table"));
```

Listing 3.4 gives an idea of how quickly data can be brought from a database-side table to a C-side matrix. The use of these structures is handled in detail in Chapter 4, so the application of the `percap` function may mystify those reading this book sequentially. But the `main` function should make sense: it opens the database, sets the `apop_opts.db_name_column` to an appropriate value, and then uses `apop_query_to_data` to pull out a data set. Its last two steps do the math and show the results on screen.

```
1  #include <apop.h>
2
3  void percap(gsl_vector *in){
4      double gdp_per_cap = gsl_vector_get(in, 1)/gsl_vector_get(in, 0);
5      gsl_vector_set(in, 2, gdp_per_cap); //column 2 is gdp_per_cap.
6  }
7
8  int main(){
9      apop_opts.verbose ++;
10     apop_db_open("data-wb.db");
11     strcpy(apop_opts.db_name_column, "country");
12     apop_data *d = apop_query_to_data("select pop.country as country, \
13         pop.population as pop, gdp.GDP as GDP, 1 as GDP_per_cap\
14         from pop, gdp \
15         where pop.country == gdp.country");
16     apop_matrix_apply(d->matrix, percap);
17     apop_data_show(d);
18     apop_opts.output_type = 'd';
19     apop_data_print(d, "wbtodata_output");
20 }
```

Listing 3.4 Query populations and GDP to an `apop_data` structure, and then calculate the GDP per capita using C routines. Online source: `wbtodata.c`.

- Line 11: As above, SQL tables have no special means of handling row names, while `apop_data` sets can have both row and column labels. You can set `apop_opts.db_name_column` to a column name that will be specially treated as holding row names for the sake of importing to an `apop_data` set.
- Lines 12–15: The final table will have three columns (pop, GDP, GDP/cap), so the query asks for three columns, one of which is filled with ones. This is known as *planning ahead*: it is difficult to resize `gsl_matrix`es and `apop_data` sets, so we query a table of the appropriate size, and then fill the column of dummy data with correct values in the C-side matrix.

Data to db To go from C-side matrices to database-side tables, there are the plain old print functions like `apop_data_print` and `apop_matrix_print`. Lines 18–19 of Listing 3.4 will write the data table to a table named `wbtodata_output`. Say that tomorrow you decide you would prefer to have the data dumped to a file; then just change the `'d'` to an `'f'` and away you go.

Crosstabs In the spreadsheet world, we often get tables in a form where both the X- and Y-dimensions are labeled, such as the case where the X-dimension is the year, the Y-dimension is the location, and the (x, y) point is a measurement taken that year at that location.

Conversely, the most convenient form for this data in a database is three columns: year, location, statistic. After all, how would you write a query such as `select statistic from tab where year < 1990` if there were a separate column for each year? Converting between the two forms is an annoyance, and so Apophenia provides functions to do conversions back and forth, `apop_db_to_crosstab` and `apop_crosstab_to_db`.

Imagine a data table with two columns, `height` and `width`, where `height` may take on values like `up`, `middle`, or `down`, and `width` takes on values like `left` and `right`. Then the query

```
create table anovatab as
    select height, width, count(*) as ct
    group by height, width
```

will produce a table looking something like

height	width	ct
up	left	12
up	right	18
middle	left	10
middle	right	7
down	left	6
down	right	18

Then, the command

```
apop_data *anova_tab = apop_db_to_crosstab("anovatab", "height", "width", "ct");
```

will put into `anova_tab` data of the form

	Left	Right
Up	12	18
Middle	10	7
Down	6	18

You can print this table as a summary, or use it to run ANOVA tests, as in Section 9.4. The `apop_crosstab_to_db` function goes in the other direction; see the online reference for details.

$\mathbb{Q}_{3.13}$ Use the command-line program `apop_db_to_crosstab` (or the corresponding C function) and the `data-climate.db` database to produce a table of temperatures, where each row is a year and each column a month. Import the output into your favorite spreadsheet program.

Multiple databases For both SQL and C, the dot means *subelement*. Just as a C `struct` named `person` might have a subelement named `person.height`, the full name of a column is `dbname.tablename.colname`.

The typical database system (including mySQL and SQLite) begins with one database open, which always has the alias `main`, but allows you to attach additional databases. For SQLite, the syntax is simply `attach database "newdb.db" as dbalias`; after this you can refer to tables via the *dbalias.tablename* form. For mySQL, you don't even need the `attach` command, and can refer to tables in other mySQL databases using the *dbname.tablename* form at any time.

Aliases again help to retain brevity. Instead of using the full *db.table.col* format for a column, this query assigns aliases for the *db.table* parts in the `from` clause, then uses those alises in the `select` clause:

```
attach database newdb as n;
select t1.c1, t2.c2
from main.firsttab t1, n.othertab t2
```

Given two attached databases, say `main` and `new`, you could easily copy tables between them via

```
create table new.tablecopy
  as select * from main.origial
```

Apophenia also provides two convenience functions, `apop_db_merge` and `apop_db_merge_table`, which facilitate such copying.

In-memory databases are faster, but at the close of the program, you may want the database on the hard drive. To get the best of both worlds, use an in-memory database for the bulk of the work, and then write the database to disk at the end of the program, e.g.:

```
int main(void){
    apop_db_open(NULL); //open a db in memory.
    do_hard_math(...);
    remove("on_disk.db");
    apop_db_merge("on_disk.db");
}
```

- `remove` is the standard C library function to delete a file.
- Removing the file before merging prevented the duplication of data (because duplicate tables are appended to, not overwritten).

- Open an SQLite database in memory using `apop_db_open(NULL)`, and on the hard drive using `apop_db_open("filename")`.
- Import data using `apop_text_to_db`.
- If you don't need output, use `apop_query` to send queries to the database engine.
- Use `apop_query_to_(data|matrix|vector|text|float)` to write a query result to various formats.

3.6 MADDENING DETAILS

Data are never as clean as it seems in the textbooks, and our faster computers have done nothing to help the fact that everybody has different rules regarding how data should be written down. Here are a few tips on dealing with some common frustrations of data importation and use; Appendix B offers a few more tools.

Spaces in column names Column names should be short and have no punctuation but underscores. Instead of a column name like `Percent of male treatment 1 cases showing only signs of nausea`, give a brief name like `male_t1_moderate`, and then create a documentation table that describes exactly what that abbreviation means.

Not everybody follows this advice, however, which creates a small frustration. The query `select 'percent of males treatment 1' from data` will produce a table with the literal string `percent of males treatment 1` repeated for each row, which is far from what you meant. The solution is to use the dot notation to specify a table: `select data.'percent of males treatment 1'`

`as males_t1 from data` will correctly return the data column, and give it an alias that is much easier to use.

Text and numbers In some cases, you need both text and numeric data in the same data set. As you will see in the next chapter, the `apop_data` structure includes slots for both text and numbers, so you only need to specify which column goes where. The first argument to the `apop_query_to_mixed_data` function is a specifier consisting of the letters `n`, `v`, `m`, `t`, indicating whether each column should be read in to the output `apop_data`'s name, vector, a matrix column, or a text column. For example, `apop_query_to_mixed_data("nmt", "select a, b*%i, c from data", counter)` would use column `a` as the row names, `b*counter` as the first column of the matrix, and `c` as a column of text elements. This provides maximal flexibility, but requires knowing exactly what the query will output.[13]

Now that you have text in an `apop_data` set, what can you do with it? In most cases, the data will be unordered discrete data, and the only thing you can do with it is to turn it into a series of dummy variables. See page 123 for an example.

Missing data Everybody represents missing data differently. SQLite uses `NULL` to indicate missing data; Section 4.5 will show that real numbers in C can take `NAN` values, whose use is facilitated by the GSL's `GSL_NAN` macro. The typical input data set indicates a missing value with a text marker like `NaN`, `..`, `-`, `-1`, `NA`, or some other arbitrary indicator.

When reading in text, you can set `apop_opts.db_nan` to a regular expression that matches the missing data marker. If you are unfamiliar with regular expressions, see Appendix B for a tutorial. For now, here are some examples:

```
//Apophenia's default NaN string, matching NaN, nan, or NAN:
strcpy(apop_opts.db_nan, "NaN");
//Literal text:
strcpy(apop_opts.db_nan, "Missing");
//Matches two periods. Periods are special in regexes, so they need backslashes.
strcpy(apop_opts.db_nan, "\\.\\.");
```

[13] Why doesn't Apophenia automatically detect the type of each column? Because it stresses replicability, and it is impossible to replicably guess column types. One common approach used by some stats packages is to look at the first row of data and use that to cast the entire column, but if the first element in a column is `NAN`, then numeric data may wind up as text or vice versa, depending on arbitrary rules. The system could search the entire column for text and presume that some count of text elements means the entire column is text, but this too is error-prone. Next month, when the new data set comes in, columns that used to be auto-typed as text may now be auto-typed as numbers, so scripts written around the first data set break. Explicitly specifying types may take work, but outguessing the system's attempts at cleaning real-world data frequently takes more work.

The searched-for text must be the entire string, plus or minus surrounding quotation marks or white space. None of these will match `NANCY` or `missing persons`.

Once the database has a `NULL` in the right place, Apophenia's functions to read between databases on one side and `gsl_matrix`es, `apop_data`, and other C structures on the other will translate between database `NULL`s and floating-point `GSL_NAN`s.

Mathematically, any operation on unknown data produces an unknown result, so you will need to do something to ensure that your data set is complete before making estimations based on the data. The naïve approach is to simply delete every observation that is not complete. Allison (2002) points out that this naïve approach, known in the jargon as *listwise deletion*, is a somewhat reasonable approach, especially if there is no reason to suspect that the pattern of missing data is correlated to the dependent variable in your study.[14] Missing data will be covered in detail on page 345.

Implementing listwise deletion in SQL is simple: given *`datacol1`* and *`datacol2`*, add a `where` *`datacol1`* `is not null and` *`datacol2`* `is not null` clause to your query. If both are numeric data, then you can even summarize this to `where (`*`datacol1`*` + `*`datacol2`*`) is not null`.

$\mathbb{Q}_{3.14}$ Using the above notes and the `data-tattoo.db` file, query to an `apop_data` set the number of tattoos, number of piercings, and the political affiliation of each subject. Make sure that all `NaN`s are converted to zeros at some point along the chain. Print the table to screen (via `apop_data_show`) to make sure that all is correctly in place. Then, query out a list of the political parties in the data set. (*Hint*: `select distinct`.) Write a `for` loop to run through the list, finding the mean number of tattoos and piercings for Democrats, Republicans, ... Would you keep the last person in the survey (who has far more tattoos than anybody else) or eliminate the person as an outlier, via a `where` clause restricting the tattoo count to under 30?

Outer join Another possibility is that a row of data is entirely missing from one table. The World Bank database includes a `lonely_planet` table listing the number of pages in the given country's Lonely Planet tourist guidebook. Antarctica has a 328-page guidebook, but no GDP and a negligible population, so the query

[14] Systematic relationships between missingness and the independent variables is much less of a concern.

```
select pp, gdp
    from lonely_planet lp, gdp
    where lp.country=gdp.country
```

will not return an Antarctica line, because there is no corresponding line in the `gdp` table. The solution is the `outer join`, which includes all data in the first table, plus data from the second table or a blank if necessary. Here is a join that will include Antarctica in its output. The condition for joining the two tables (join `on l.country=gdp.country`) now appears in a different location from the norm, because the entire left outer join clause describes a single table to be used as a data source.

```
select pp, gdp
    from lonely_planet lp left outer join gdp
            on l.country=gdp.country
    where l.country like 'A%'
```

Q 3.15 The query above is a *left outer join*, which includes all data from the left table, but may exclude data from the right table. As of this writing, this is all that SQLite supports, but other systems also support the *right outer join* (include all entries in the right table) and the *full outer join* (include all entries from both tables).
Using the `union` keyword, generate a reference table with all of the country names from both the Lonely Planet and GDP tables. Then use a few left outer joins beginning with the reference table to produce a complete data set.

※ *mySQL* As well as SQLite, Apophenia supports mySQL. mySQL is somewhat better for massive data sets, but will work only if you already have a mySQL server running, have permission to access it, and have a database in place. Your package manager will make installing the mySQL server, client, and development libraries easy, and mySQL's maintainers have placed online a comprehensive manual with tutorial.

Once mySQL is set up on your system, you will need to make one of two changes: either set your shell's `APOP_DB_ENGINE` environment variable to `mysql`,[15] or in your code, set `apop_opts.db_engine='m'`. You can thus switch back and forth between SQLite and mySQL; if the variable is `'m'` then any database operations will go to the mySQL engine and if it is not, then database operations will be sent

[15] As discussed in Appendix A, you will probably want to add `export APOP_DB_ENGINE=mysql` to your `.bashrc` on systems using `mySQL`.

to the SQLite engine. This could be useful for transferring data between the two. For example:

```
apop_opts.db_engine = 'm';
apop_db_open("mysqldb");
apop_data *d = apop_query_to_data("select * from get_me");
apop_opts.db_engine = 'l';
apop_db_open("sqlitedb");
apop_opts.output_type = 'd'; //print to database.
apop_data_print(d, "put_me");
```

SQLite's concept of a database is a single file on the hard drive, or a database in memory. Conversely mySQL has a server that stores all databases in a central repository (whose location is of no concern to end-users). It has no concept of an in-memory database.

As noted above, every SQL system has its own rules for metatadata. From the `mysql` prompt, you can query the mySQL server for a complete list of databases with `show databases`, and then attach to one using `use` *`dbname`*; (or type `mysql` *`dbname`* at the command prompt to attach to *`dbname`* from the outset). You can use `show tables;` to get the list of tables in the current database (like the SQLite prompt's `.tables` command), or use `show tables from your_db;` to see the tables in `your_db` without first attaching to it. Given a table, you can use `show columns from` *`your_table`* to see the column names of `your_table`.[16]

mySQL digresses from the SQL standard in different manners from SQLite's means of digressing from the standard:

- SQLite is somewhat forgiving about details of punctuation, such as taking `==` and `=` as equivalent, and "double-ticks" and 'single-ticks' as equivalent. mySQL demands a single `=` and 'single-ticks'.
- After every `select`, `create`, and so on, mySQL's results need to be internally processed, lest you get an error about commands executed out of order. Apophenia's functions handle the processing for you, but you may still see odd effects when sending a string holding multiple semicolon-separated queries to the `apop_query...` functions. Similarly, you may have trouble using `begin/commit` wrappers to bundle queries, though mySQL's internal cache management may make such wrappers unnecessary.
- mySQL includes many more functions beyond the SQL standard, and has a number of additional utilities. For example, there is a `LOAD` command that will read in a text file much more quickly than `apop_text_to_db`.

[16] Or, use the command-line program `mysqlshow` to do all of these things in a slightly more pleasant format.

- SQL represents missing data via a `NULL` marker, so queries may include conditions like `where col is not null`.
- Data files use whatever came to mind to mark missing data, so set `apop_opts.db_nan` to a regular expression appropriate for your data.
- If a name appears in one table but not another, and you would like to joint tables by name, use the `outer join` to ensure that all names appear.

3.7 SOME EXAMPLES

Here are a few examples of how C code and SQL calls can neatly interact.

TAKING SIMULATION NOTES Say that you are running a simulation and would like to take notes on its state each period. The following code will open a file on the hard drive, create a table, and add an entry each period. The begin-commit wrapper puts data in chunks of 10,000 elements, so if you get tired of waiting, you can halt the program and walk away with your data to that point.[17]

```
double sim_output;
apop_db_open("sim.db");
apop_table_exists("results", 1); //See below.
apop_query("create table results (period, output); begin;");
for (int i=0; i< max_periods; i++){
    sim_output = run_sim(i);
    apop_query("insert into results values(%i, %g);", i, sim_output);
    if (!(i%1e4))
        apop_query("commit; begin;");
}
apop_query("commit;");
apop_db_close(0);
```

- The `apop_table_exists` command checks for whether a table already exists. If the second argument is one, as in the example above, then the table is deleted so that it can be created anew subsequently; if the second argument is zero, then the function simply returns the answer to the question "does the table exist?" but leaves the table intact if it is there. It is especially useful in `if` statements.
- Every `1e4` entries, the system commits what has been entered so far and begins a new batch. With some SQLite systems, this can add significant speed. mySQL

[17] Sometimes such behavior will leave the database in an unclean state. If so, try the SQLite command `vacuum`.

does its own batch management, so the `begin`s and `commit`s should be omitted for mySQL databases.

EASY T-TESTS People on the East and West coasts of the United States sometimes joke that they can't tell the difference between all those states in the middle. This is a perfect chance for a t test: are incomes in North Dakota significantly different from incomes in South Dakota? First, we will go through the test algorithm in English, and then see how it is done in code.

Let the first data set be the income of counties in North Dakota, and let the second be the income of counties in South Dakota. If $\hat{\mu}$, $\hat{\sigma}^2$, and n are the estimated mean, variance, and actual count of elements of the North and South data sets,

$$\texttt{stat} = \frac{\hat{\mu}_N - \hat{\mu}_S}{\sqrt{\hat{\sigma}_N^2/n_N + \hat{\sigma}_S^2/n_S}} \sim t_{n_N+n_S-2}. \tag{3.7.1}$$

[That is, the given ratio has a t distribution with $n_N + n_S - 2$ degrees of freedom.]

The final step is to look up this statistic in the standard t tables as found in the back of any standard statistics textbook. Of course, looking up data is the job of a computer, so we instead ask the GSL for the two-tailed confidence level (see page 305 for details):

```
double confidence = (1 - 2* gsl_cdf_tdist_Q(|statl, n_N + n_S -2));
```

If `confidence` is large, say $> 95\%$, then we can reject the null hypothesis that North and South Dakotan incomes (by county) are different. Otherwise, there isn't enough information to say much with confidence.

Listing 3.5 translates the process into C.

- Lines 4–8 comprise two queries, that are read into a `gsl_vector`. Both ask for the same data, but one has a `where` clause restricting the query to pull only North Dakotan counties, and the other has a `where` clause restricting the query to South Dakota.
- Lines 10–15 get the vital statistics from the vectors: count, mean, and variance.
- Given this, line 17 is the translation of Equation 3.7.1.
- Finally, line 18 is the confidence calculation from above, which line 19 prints as a percentage.

```
1  #include <apop.h>
2
3  int main(){
4      apop_db_open("data-census.db");
5      gsl_vector *n = apop_query_to_vector("select in_per_capita from income "
6                  "where state= (select state from geography where name ='North Dakota')");
7      gsl_vector *s = apop_query_to_vector("select in_per_capita from income "
8                  "where state= (select state from geography where name ='South Dakota')");
9
10     double n_count = n->size,
11                 n_mean = apop_vector_mean(n),
12                 n_var = apop_vector_var(n),
13                 s_count = s->size,
14                 s_mean = apop_vector_mean(s),
15                 s_var = apop_vector_var(s);
16
17     double stat = fabs(n_mean - s_mean)/ sqrt(n_var/ (n_count-1) + s_var/(s_count-1));
18     double confidence = 1 - (2 * gsl_cdf_tdist_Q(stat, n_count + s_count -2));
19     printf("Reject the null with %g%% confidence\n", confidence*100);
20 }
```

Listing 3.5 Are North Dakota incomes different from South Dakota incomes? Answering the long way. Online source: `ttest.long.c`.

No, easier But this is not quite as easy as it could be, because Apophenia provides a high-level function to do the math for you, as per Listing 3.6. The code is identical until line eight, but then line nine calls the `apop_t_test` function, which takes the two vectors as input, and returns an `apop_data` structure as output, listing the relevant statistics. Line ten prints the entire output structure, and line eleven selects the single confidence statistic regarding the two-tailed hypothesis that $\text{income}_{\text{ND}} \neq \text{income}_{\text{SD}}$.

DUMMY VARIABLES The `case` command is the if-then-else of SQL. Say that you have data that are true/false or yes/no. One way to turn this into a one-zero variable would be via the `apop_data_to_dummies` function on the matrix side. This works partly because of our luck that $y > n$ and $T > F$ in English, so y and T will map to one and n and F will map to zero. But say that our survey used *affirmative* and *negative*, so the mapping would be backward from our intuition. Then we can put a `case` statement in with the other column definitions to produce a column that is one when `binaryq` is affirmative and zero otherwise:

```
select id,
    case binaryq when "affirmative" then 1 else 0 end,
    other_vars
from datatable;
```

```
1  #include <apop.h>
2
3  int main(){
4      apop_db_open("data-census.db");
5      gsl_vector *n = apop_query_to_vector("select in_per_capita from income "
6              "where state= (select state from geography where name ='North Dakota')");
7      gsl_vector *s = apop_query_to_vector("select in_per_capita from income "
8              "where state= (select state from geography where name ='South Dakota')");
9      apop_data *t = apop_t_test(n,s);
10     apop_data_show(t); //show the whole output set...
11     printf ("\n confidence: %g\n", apop_data_get_ti(t, "conf.*2 tail", -1)); //...or just one value.
12 }
```

Listing 3.6 Are North Dakota incomes different from South Dakota incomes? Online source: `ttest.c`.

To take this to the extreme, we can turn a variable that is discrete but not ordered (such as district numbers in the following example) into a series of dummy variables. It requires writing down a separate `case` statement for each value the variable could take, but that's what `for` loops are for. [Again, this is demonstration code. Use `apop_data_to_dummies` to do this in practice.] Listing 3.7 creates a series of dummy variables using this technique.

- On lines 5–6, the `build_a_query` function queries out the list of districts.
- Then the query writes a select statement with a line `case State when` *`state_name`* `then 1 else 0` for every *`state_name`*.
- Line 11 uses the obfuscatory if (page 211) to print a comma between items, but not at the end of the `select` clause.
- Line 18 pulls the data from this massive query, and line 19 runs an OLS regression on the returned data.
- You can set `apop_opts.verbose=1` at the head of `main` to have the function display the full query as it executes.
- Lines 20–21 show the parameter estimates, but suppress the gigantic variance–covariance matrix.

Note well that the `for` loop starting on line eight goes from `i=1`, not `i=0`. When including dummy variables, you always have to exclude one baseline value to prevent $\mathbf{X}$ from being singular; excluding `i=0` means Alabama will be the baseline. $\mathbb{Q}$: Rewrite the `for` loop to use another state as a baseline. Or, set the `for` loop to run the full range from zero to the end of the array, and watch disaster befall the analysis.

```
1  #include <apop.h>
2
3  char *build_a_query(){
4     char *q = NULL;
5        apop_data *state = apop_query_to_text("select Name as state, State as id \
6                        from geography where sumlevel+0.0 = 40");
7        asprintf(&q, "select in_per_capita as income, ");
8        for (int i=1; i< state->textsize[0]; i++)
9            asprintf(&q, "%s (case state when '%s' then 1 else 0 end) '%s' %c \n",
10                           q, state->text[i][1], state->text[i][0],
11                           (i< state->textsize[0]-1) ? ',':' ');
12       asprintf(&q,"%s from income\n", q);
13       return q;
14 }
15
16 int main(){
17       apop_db_open("data-census.db");
18       apop_data *d = apop_query_to_data(build_a_query());
19       apop_model *e = apop_estimate(d, apop_ols);
20       e->covariance = NULL; //don't show it.
21       apop_model_show(e);
22 }
```

Listing 3.7 A sample of a `for` loop that creates SQL that creates dummy variables. Online source: `statedummies.c`.

- There is no standard for `for` loops, assigning variables, or matrix-style manipulation within SQL, so you need to do these things on the C-side of your analysis.
- Functions exist to transfer data between databases and matrices, so you can incorporate database-side queries directly into C code.