Relational Databases and SQL

A *database* is a collection of data with information about how the data are organized (meta-data). A *database server* is like a web server, but responds to requests for data rather than web pages.

We'll talk about *relational database management systems* (RDBMS) and how to communicate with them using the *structured query language* (SQL).

Why use a database?

- Coordinate synchronized access to data
- Change continually; give immediate access to live data
- Centralize data for backups
- Control access to the data

A RDBMS had three main parts

- Data definition
- Data access
- Privilege management

We will concentrate on data access, assuming the database is already available and we have the needed privileges.

Topics:

- use SQL to extract information from an RDBMS
- relate these SQL statements back to similar tasks in R
- use SQL from within R

There are tradeoffs in terms of what we choose to do using SQL and what we do in R.

- A database is made up of one or more two dimensional *tables*, usually stored as files on the server.
- A *table* is a rectangular arrangement of values, where rows represent cases, and columns represent variables (just like data frames in R).
- Another term for a table is a *relation*. Rows are also referred to as *tuples* and the columns as *attributes*.

- SQL allows us to interactively *query* the database to reduce the data by subsetting, grouping, or aggregation.
- Each database program tends to have its own version of SQL, but they all support the same basic SQL statements. (We say statements rather than commands because SQL is referred to as a declarative rather than an imperative language.)
- The SQL statement for retrieving data is the SELECT statement. This operates on one or more tables. The result will always be another table.

We have a table called chips, with data about the CPU development of PCs over time

The simplest possible query gives back everything:

```
SELECT * FROM chips;
```

```
 processor   | date | transistors  | microns | clockspeed  | width | mips
-------------+------+--------------+---------+-------------+-------+------
 8080        | 1974 |         6000 |       6 |           2 |     8 | 0.64
 8088        | 1979 |        29000 |       3 |           5 |    16 | 0.33
 80286       | 1982 |       134000 |     1.5 |           6 |    16 |    1
 80386       | 1985 |       275000 |     1.5 |          16 |    32 |    5
 80486       | 1989 |      1200000 |       1 |          25 |    32 |   20
 Pentium     | 1993 |      3100000 |     0.8 |          60 |    32 |  100
 PentiumII   | 1997 |      7500000 |    0.35 |         233 |    32 |  300
 PentiumIII  | 1999 |      9500000 |    0.25 |         450 |    32 |  510
 Pentium4    | 2000 |     42000000 |    0.18 |        1500 |    32 | 1700
```

- By convention, we display SQL statements in upper case. Statements must end with a semicolon.

Attributes / Variables

- Recall that in R, we can select particular variables (columns) by name.

```
chips[ , c('mips', 'microns')]
```

- The order of the variable names determines the order in which they are returned in the resulting data frame.
- The corresponding SQL query is

```
SELECT mips, microns FROM chips;
```

SQL Syntax

- Similar to a sentence in English, except that there's less flexibility in the order of the words.
- Sentence ends with a ;
- Use blanks and "," and "=" and "()" as delimiters
- We will only look at SELECT statements, i.e., statements that begin with the term SELECT

Examples of SELECT statements

```
SELECT * FROM chips;
SELECT mips, microns FROM chips;
```

```
SELECT * FROM chips
 WHERE processor = ‘Pentium’ OR
    processor = ‘PentiumII’;
```

- In R, it usually does not matter whether you use single or double quotes to surround character strings. In SQL, the standard is to use single quotes so we will do this throughout for both R and SQL.
- The `WHERE` clause can also be used with Boolean operators. The keyword NOT negates a condition, and parentheses can be used to clarify order of evaluation.

```
SELECT * FROM chips WHERE date > 1990;

SELECT * FROM chips WHERE NOT width = 8;

SELECT * FROM chips WHERE NOT (width = 8 OR
width = 16);
```

Selecting Tuple/Row

In R we can select rows that match a condition:

```
chips[chips$processor == ‘Pentium’ |
      chips$processor == ‘PentiumII’, ]
```

The corresponding SQL statement is

```
SELECT * FROM chips
 WHERE processor = ‘Pentium’ OR
   processor = ‘PentiumII’;
```

Note: Whitespace can be used freely in SQL statements. We often separate and indent lines for clarity. The statement isn’t evaluated until the semicolon is entered.

In both R and SQL, we can do both types of subsetting at once, i.e., subset columns and rows

In R:

```
chips[chips$processor == ‘Pentium’ |
       chips$processor == ‘PentiumII’,
      c(‘mips’, ‘microns’)]
```

In SQL:

```
SELECT mips, microns FROM chips
    WHERE processor = ‘Pentium’ OR
          processor = ‘PentiumII’;
```

General Syntax

SELECT attribute(s) FROM table(s)
[WHERE constraints]; ← [optional]

How would we pull the years of all 32-bit processors that execute fewer than 250 million instructions per second (mips),

In R,

```
chips[chips$mips < 250 & chips$width == 32, "date"]
```

In SQL

```
SELECT date FROM chips
  WHERE mips < 250 AND width = 32;
```

SQL offers limited features for summarizing data -- some aggregate functions that operate over the rows of a table and some mathematical functions that operate on individual values in a row.

The aggregate functions are

- COUNT - number of rows
- SUM - total of all values for an attribute
- AVG - average value for an attribute
- MIN - minimum value for an attribute
- MAX - maximum value for an attribute

SELECT attribute(s) FROM table(s) [WHERE constraints];

↖ can also be aggregate *functions* of attributes

Additional clauses: GROUP BY

- The GROUP BY clause makes the aggregate functions in SQL more useful. It enables the aggregates to be applied to *subsets* of the rows in a table.

```
SELECT width, MAX(mips) FROM chips
    GROUP BY width;
```

```
 width | max
-------+------
     8 | 0.64
    16 |    1
    32 | 1700
(3 rows)
```

- More than one attribute can be included in the GROUP BY clause.

Additional clauses: HAVING

- The WHERE clause can't contain an aggregate function, but the HAVING clause can be used to refer to the groups to be selected.

```
SELECT width, MAX(mips) FROM chips
    GROUP BY width HAVING MAX(mips) >= 1;
```

```
 width | max
-------+------
    16 |    1
    32 | 1700
(2 rows)
```

- First the chips table is separated into sets of rows by width. For each set, MAX(mips) is calculated, and the set is discarded if MAX(mips) < 1. Finally, width and MAX(mips) are returned for each set.

A few other predicates and clauses

- DISTINCT - forces values of an attribute in the results table to have unique values
- NOT - negates conditions in WHERE or HAVING clause
- LIMIT - limits the number of rows returned

```
SELECT * FROM chips LIMIT 3;

SELECT DISTINCT width FROM chips;
```

Order of Execution

The order of execution of the clauses in a `SELECT` statement is as follows:

1. `FROM:` The working table is constructed.

2. `WHERE:` The `WHERE` clause is applied to each row of the table, and only the rows that test TRUE are retained.

3. `GROUP BY:` The results are broken into groups of rows all with the same value of the `GROUP BY` clause.

4. `HAVING`: The `HAVING` clause is applied to each group and only those that test `TRUE` are retained.

5. `SELECT:` The attributes not in the list are dropped, aggregates are calculated, and options `DISTINCT`, `ORDER BY` and `LIMIT` are applied.

Exercises:

1) How many rows are in the chips table?

```
SELECT COUNT(*) FROM chips;
```

2) How many chips have attribute width equal to 32?

```
SELECT COUNT(*) FROM chips WHERE width = 32;
```

3) What is the average clock speed for the chips in question 2?

```
SELECT AVG(clockspeed) FROM chips WHERE width = 32;
```

Now answer the same questions assuming chips is a data frame in R.

```
nrow(chips);
sum(chips$width == 32);
mean(chips$clockspeed[chips$width == 32)
```

Exercises:

4) What is the average clock speed for each unique value of width?

```
SELECT AVG(clockspeed) FROM chips
    GROUP BY width;
```

5) How many chips are in each width group (except width 8)?

```
SELECT COUNT(*) FROM chips
GROUP BY width HAVING width > 8;
```

Now answer the same questions assuming chips is a data frame in R.

```
tapply(chips$clockspeed,
       chips$width, mean)
```

Exercises:

6) For those chips models developed before 2000, what is the average clock speed per width?

```
A) SELECT AVG(clockspeed) FROM chips
          GROUP BY width
          HAVING date < 2000;

B) SELECT AVG(clockspeed) FROM chips
          WHERE date < 2000
          GROUP BY width;
```

Now answer the same questions assuming chips is a data frame in R.

```
with(chips[chips$date < 2000],
     tapply(clockspeed, width, mean)
```

Using SQL with R

The DBI, RSQLite libraries in R allow you to connect to an SQL database, submit a query, and receive the results as a data frame.

```
library(RSQLite)
```

Set up an interface to SQLite

```
drv = dbDriver("SQLite")
```

Connect to DBMS (no security SQLite)

```
con = dbConnect(drv,
                dbname="chipsDB")
```

Using SQL with R

The RMySQL library also allows you to connect to a mySQL database.

```
library(RMySQL)
```

Set up an interface to MySQL

```
drv = dbDriver("MySQL")
```

Connect to DBMS with security

```
con = dbConnect(drv, dbname = "lahman",
  user = "s133", password = "s133",
  host = "radagast.berkeley.edu")
```

Sending an SQL query to the database with R

A data frame is returned

```
X = dbGetQuery(con,
          "SELECT * FROM chipsSQLite LIMIT 5;")

class(X)
[1] "data.frame"

dim(X)
[1] 5 7
```

We can of course import the whole table, then extracting what we want using R commands.

But, it may be more efficient to use SELECT to extract and import only the results with which we want to work

If the results are large, we don't have to pull them all over into R at once.

Submit an SQL statement; keep results in database
Use dbSendQuery to do this rather than dbGetQuery

```
rs = dbSendQuery(con,
                  "SELECT * FROM
chipsSQLite;")
```

Retrieve the first 5 rows in the results

```
fetch(rs, n = 5)
```

Retrieve the next 3 rows

```
fetch(rs, n = 3)
```

Retrieve the remaining rows

```
fetch(rs, n = -1)
```

Close the query

```
dbClearResult(rs)
```

Can close without retrieving all rows

Close the connection and turn off the driver

Disconnect from the database.

```
dbDisconnect(con)
[1] TRUE
```

Unload the Driver

```
dbUnloadDriver(drv)
[1] TRUE
```