Lecture 25: Database Notes

36-350, Fall 2012*

November 30, 2012

The examples here use `http://www.stat.cmu.edu/~cshalizi/statcomp/labs/baseball.db`, which is derived from Lahman's baseball database (`http://www.seanlahman.com/baseball-archive/statistics/`), more fully documented at `http://seanlahman.com/files/database/readme58.txt`.

A database consists of one or more tables; each table is like a data-frame in R, where each record is like a data-frame's row, and each field is like a column. Related tables are linked by the use of unique identifiers or keys (.e.g., patient IDs).

A database server hosts one or more databases. The server controls access to the databases, and handles the actual business of reading out information from the tables efficiently (or writing information to them). Database clients are programs which interact with the server. (Think of the difference between the Web browser program on your computer or phone, and the Web server program which actually hosts the website.)

The main thing you will be doing with a database is issuing queries — asking it to retrieve selected parts of a database. Just as we used regular expressions to specify patterns of text, we need a special syntax to specify queries unambiguously. The *de facto* standard is the Structured Query Language, SQL. Most popular database server programs are designed to work with it, sometimes with slight variants from program to program.

*With thanks to Vince Vu

Common Operations

	SQL command	SQLite variant
List available databases	`SHOW DATABASES`	`.databases`
List tables in a database	`SHOW TABLES IN` database	`.tables`
List fields in a table	`SHOW COLUMNS IN` table	
Describe the types of the columns in a table	`DESCRIBE` table	`.schema` table
Change the default database	`USE` database	

Select

The main tool is the SELECT command: you give it a specification of what information you want, and it returns a table:

```
SELECT columns or computations
  FROM table
  WHERE condition
  GROUP BY columns
  HAVING condition
  ORDER BY column [ASC|DESC]
  LIMIT offset,count;
```

Most of this is optional (but not the semi-colon at the end).

0.1 Selecting Columns

Columns get specified by name (which are case-sensitive), with multiple columns separated by commas. `*` specifies all columns.

```
SELECT PlayerID,yearID,AB,H FROM Batting;
```

returns a sub-table, with the specified columns, from the table `Batting`.

```
SELECT * FROM Salaries;
```

returns all columns from the table `Salaries`.

```
SELECT * FROM Salaries ORDER BY Salary;
```

re-organizes so the records are sorted by the `Salary` field. The default is ascending order.

```
SELECT * FROM Salaries ORDER BY Salary DESC;
```

reverses the order.

```
SELECT * FROM Salaries ORDER BY Salary DESC LIMIT 10;
```

will return all columns for the records with the 10 highest salaries.

0.2 Selecting Records

SQL uses Boolean expressions in the WHERE clause to decide which records get selected, like `subset()` or `which()` in R.

```
SELECT PlayerID,yearID,AB,H FROM Batting WHERE AB > 100 AND H > 0;
```

will return the specified columns from `Batting`, but only for the records where the `AB` field is over 100 and the `H` field is over 0.

0.3 Calculated Columns

We can have the server do some calculations for us as part of the query, including simple arithmetic and basic summaries, like `SUM`, `AVG`, `MIN`, `MAX`, `COUNT`, `VAR_SAMP`, `STDDEV_SAMP`.

```
SELECT MAX(AB) FROM Batting;
SELECT MIN(AB), AVG(AB), MAX(AB) FROM Batting;
```

do what you'd expected.

```
SELECT AB, H, H/AB FROM Batting;
```

doesn't do quite what you expect — since `H` and `AB` are both integers, `H/AB` is just the integer part of the ratio.

```
SELECT AB,H,H/CAST(AB AS REAL) FROM Batting;
```

gives the correct floating-point ratio.

We can give calculated columns names, if we want to refer to them, e.g., for sorting:

```
SELECT PlayerID,yearID,H/CAST(AB AS REAL) AS BattingAvg FROM Batting
  ORDER BY BattingAvg DESC LIMIT 10;
```

returns the records with the 10 highest values of our new field `BattingAvg`.

0.4 Aggregation

To aggregate, i.e., to do calculations on grouped subsets of records, we use `GROUP BY` clauses, much like `dply` in R.

```
SELECT playerID, SUM(salary) FROM Salaries GROUP BY playerID
```

gives the total salary paid over time to each player.

These calculations can be named:

```
SELECT playerID, SUM(salary) AS totalSalary FROM Salaries GROUP BY playerID
  ORDER BY totalSalary DESC LIMIT 10;
```

Exercise: What would

```
SELECT salary, COUNT(playerID) FROM Salaries GROUP BY salary
```

do?

0.5 Selecting Records with Aggregation

SELECT processes part of what you tell it to do in order: WHERE clauses come first (initial selection of records), then GROUP BY (aggregation of records), then HAVING for post-aggregation selection.

```
SELECT playerID, SUM(salary) AS totalSalary FROM Salaries GROUP BY playerID
  HAVING totalSalary > 200000000
```

will return grouped records (i.e., players) where the total salary exceeds 200 million.

JOIN

If information is split across tables, we need to join it back together. The operation which does so is called a "join", naturally enough, which in SQL shows up as a JOIN clause in SELECT.

The most natural kind of JOIN is where we find records in two tables with matching keys (unique identifiers), and merge the sets of fields.

(More generally, we look at every combination of a record from table A with a record from table B, and ask whether the pair satisfies some "JOIN condition" or "JOIN predicate"; if it does, the pair goes in to the new table as a single record. In the obvious sort of JOIN, the condition is "do the keys match?")

```
SELECT nameLast,nameFirst,yearID,AB,H FROM Master INNER JOIN Batting
  USING(playerID);
```

The INNER JOIN ... USING part creates a (virtual) table, merging the fields from `Master` and `Batting` where `playerID` matches.

An equivalent:

```
SELECT nameLast,nameFirst,yearID,AB,H FROM Master INNER JOIN Batting
  ON Master.playerID == Batting.playerID;
```

This form can be used when the key has different names in different tables.

If there is just one field in common between two tables, we can write NATURAL JOIN instead of INNER JOIN, and skip USING or ON.

```
SELECT nameLast,nameFirst,yearID,AB,H FROM Master NATURAL JOIN Batting;
```

A single SELECT can contain multiple JOINs, to combine more than two tables:

```
SELECT nameLast,nameFirst,yearID,AB,H,salary
  FROM Master NATURAL JOIN Batting NATURAL JOIN Salaries
  ORDER BY salary DESC LIMIT 10;
```

Two tables may have columns with the same name which we want to keep track of separately; then the SQL convention is to refer to them as `table.var`. The AS command lets us rename them inside a SELECT:

```
SELECT patient.last_name AS patname,
  physicians.last_name AS docname
  FROM patients NATURAL JOIN physicians;
```

AS can also be used to abbreviate or rename tables:

```
SELECT p.last_name AS patname,
  d.last_name AS docname
  FROM patients AS p NATURAL JOIN physicians AS d;
```

Accessing Databases from R

Each database management system (DBMS) has a somewhat different protocol for actually dealing with servers. Rather than trying to master all of them, the sensible approach in R is to use the `DBI` package, which provides a single uniform interface, no matter what the DBMS. The programmers of DBI have to figure out how to deal with them, but you don't. The sub-programs which handle different DBMS's are called drivers; see `http://cran.r-project.org/package=DBI`.

```
install.packages("DBI", dependencies = TRUE) # Install DBI
install.packages("RSQLite", dependencies = TRUE) # Install driver for SQLite
```

R uses persistent objects called "connections" to keep track of which database server is being used for what task.[1] So the procedure goes like so:

- Load the driver for the DBMS you'll be dealing with.
- Establish a connection to the database server.
- Interact with the DBMS.
- Close the connection.
- Unload the driver.

Load Driver, Establish Connection

```
library(RSQLite)
drv <- dbDriver('SQLite')
con <- dbConnect(drv, dbname="baseball.db")
```

`con` now is the name for the connection to the SQLite database `baseball.db`. We could also give `dbConnect` arguments `host` (an Internet address), `use` (a user name) and `password`.

Interacting with a Database through DBI

The basic commands:

```
dbListTables(conn)         # Get tables in the database (returns vector)
dbListFields(conn, name)  # List fields in a table
dbReadTable(conn, name)   # Import a table as a data frame
```

The really useful one:

```
dbGetQuery(conn, statement)
```

[1] Connections actually get used for other things as well, like file input/output.

The second argument is an SQL query, which gets issued to the database; it returns a data frame. When building the statement, as when building a regular expression, it often helps to use `paste`:

```
df <- dbGetQuery(con, paste(
  "SELECT nameLast,nameFirst,yearID,salary",
  "FROM Master NATURAL JOIN Salaries"))
```

Disconnecting and unloding the driver

There's an overhead involved in keeping these around, so remove them once they're not needed:

```
dbDisconnect(con)
dbUnloadDriver(drv)
```