Lecture 23: Databases

36-350
12 November 2014

Agenda

• What databases are, and why
• SQL
• Interfacing R and SQL

Reading: Spector, chapter 3; handout on class website

Databases

• A record is a collection of fields
• A table is a collection of records which all have the same fields (with different values)
• A database is a collection of tables

Databases vs. Dataframes

• R’s dataframes are actually tables

<table>
<thead>
<tr>
<th>R jargon</th>
<th>Database jargon</th>
</tr>
</thead>
<tbody>
<tr>
<td>column</td>
<td>field</td>
</tr>
<tr>
<td>row</td>
<td>record</td>
</tr>
<tr>
<td>dataframe</td>
<td>table</td>
</tr>
<tr>
<td>types of the columns</td>
<td>table schema</td>
</tr>
<tr>
<td>bunch of related dataframes</td>
<td>database</td>
</tr>
</tbody>
</table>

So, Why Do We Need Database Software?

• Size
  – R keeps its dataframes in memory
  – Industrial databases can be much bigger
  – Work with selected subsets

• Speed
  – Clever people have worked very hard on getting just what you want fast

• Concurrency
  – Many users accessing the same database simultaneously
  – Lots of potential for trouble (two users want to change the same record at once)
The Client-Server Model

- Databases live on a **server**, which manages them
- Users interact with the server through a **client** program
- Lets multiple users access the same database simultaneously

SQL

- **SQL** (structured query language) is the standard for database software
- Mostly about **queries**, which are like doing a selection in R

```r
debt[debtx$Country=="France",c("growth","ratio")]
with(debt,debtx[debtx$Country=="France",c("growth","ratio")])
subset(x=debtx,subset=(debtx$Country=="France"),select=c("growth","ratio"))
```

- Let’s look at how SQL does stuff like this

**SELECT**

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

- **SELECT** is the first word of a query, then modifiers say which fields/columns to use, and what conditions records/rows must meet, from which tables
- The final semi-colon is obligatory

**SELECT**

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

Four columns from table Batting

```sql
SELECT * FROM Salaries;
```

All columns from table Salaries

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

As above, but by ascending value of Salary

```sql
SELECT * FROM Salaries ORDER BY Salary DESC;
```
Descending order

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

top 10 salaries

**SELECT**

Picking out rows meeting a condition

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

vs.

```r
Batting[Batting$AB>100 & Batting$H > 0, c("PlayerID","yearID","AB","H")]
```

**Calculated Columns**

- SQL knows about some simple summary statistics:

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

- It can do arithmetic

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

  *Because `AB` and `H` are integers, and it won't give you a fractional part by default*

- Calculated columns can get names:

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

**Aggregating**

We can do calculations on value-grouped subsets, like in `aggregate` or `dply`

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

**Selecting Again**

- First cut of records is with `WHERE`
- Aggregation of records with `GROUP BY`
- Post-aggregation selection with `HAVING`

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

• So far FROM has just been one table
• Sometimes we need to combine information from many tables

<table>
<thead>
<tr>
<th>patient_last</th>
<th>patient_first</th>
<th>physician_id</th>
<th>complaint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Morgan</td>
<td>Dexter</td>
<td>37010</td>
<td>insomnia</td>
</tr>
<tr>
<td>Soprano</td>
<td>Anthony</td>
<td>79676</td>
<td>malaise</td>
</tr>
<tr>
<td>Swearengen</td>
<td>Albert</td>
<td>NA</td>
<td>healthy as a goddam horse</td>
</tr>
<tr>
<td>Garrett</td>
<td>Alma</td>
<td>90091</td>
<td>nerves</td>
</tr>
<tr>
<td>Holmes</td>
<td>Sherlock</td>
<td>43675</td>
<td>nicotine-patch addiction</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>physician_last</th>
<th>physician_first</th>
<th>physicianID</th>
<th>plan</th>
</tr>
</thead>
<tbody>
<tr>
<td>Meridian</td>
<td>Emmett</td>
<td>37010</td>
<td>UPMC</td>
</tr>
<tr>
<td>Melfi</td>
<td>Jennifer</td>
<td>79676</td>
<td>BCBS</td>
</tr>
<tr>
<td>Cochran</td>
<td>Amos</td>
<td>90091</td>
<td>UPMC</td>
</tr>
<tr>
<td>Watson</td>
<td>John</td>
<td>43675</td>
<td>VA</td>
</tr>
</tbody>
</table>

JOIN

• Suppose we want to know which doctors are treating patients for insomnia
• Complaints are in one table
• Physicians are in the other
• In R, we’d use `merge` to link the tables up by `physicianID`
• Here, `physician_id` or `physicianID` is acting as the key or unique identifier

JOIN

• SQL doesn’t have `merge`, it has JOIN as a modifier to FROM

```
SELECT physician_first, physician_last FROM patients
INNER JOIN physicians
ON patients.physician_id == physicians.physicianID
WHERE condition == "insomnia"
```

Creates a (virtual) table linking records where `physician_id` in one table matches `physicianID` in the other

• If the names were the same in the two tables, we could write (e.g.)

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

INNER JOIN ... USING links records with the same value of `playerID`

• There are some syntax variants here; see the handout
JOIN

- **LEFT OUTER JOIN** includes records from the first table which don’t match any record in the 2nd
  - The “extra” records get NA in the 2nd table’s fields
- **RIGHT OUTER JOIN** is just what you’d think
  - so is **FULL OUTER JOIN**

Updated Translation Table

<table>
<thead>
<tr>
<th>R jargon</th>
<th>Database jargon</th>
</tr>
</thead>
<tbody>
<tr>
<td>column</td>
<td>field</td>
</tr>
<tr>
<td>row</td>
<td>record</td>
</tr>
<tr>
<td>dataframe</td>
<td>table</td>
</tr>
<tr>
<td>types of the columns</td>
<td>table schema</td>
</tr>
<tr>
<td>bunch of dataframes</td>
<td>database</td>
</tr>
<tr>
<td>selections, subset</td>
<td>SELECT ... FROM ... WHERE ... HAVING</td>
</tr>
<tr>
<td>aggregate, d*ply</td>
<td>GROUP BY</td>
</tr>
<tr>
<td>merge</td>
<td>JOIN</td>
</tr>
<tr>
<td>order</td>
<td>ORDER BY</td>
</tr>
</tbody>
</table>

Connecting R to SQL

- SQL is a language; database management systems (DMBS) actually implement it and do the work
  - MySQL, SQLite, etc., etc.
- They all have somewhat different conventions
- The R package DBI is a unified interface to them
- Need a separate “driver” for each DBMS

Connecting R to SQL

```r
install.packages("DBI", dependencies = TRUE) # Install DBI
install.packages("RSQLite", dependencies = TRUE) # Install driver for SQLite
library(RSQLite)
drv <- dbDriver('SQLite')
con <- dbConnect(drv, dbname="baseball.db")

con is now a persistent connection to the database baseball.db
```

Connecting R to SQL

```r
dbListTables(con) # Get tables in the database (returns vector)
dbListFields(con, name) # List fields in a table
dbReadTable(con, name) # Import a table as a data frame
```
Connecting R to SQL

dbGetQuery(conn, statement)

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

Connecting R to SQL

Usual workflow: - Load the driver, connect to the right database - R sends an SQL query to the DBMS - SQL executes the query, sending back a manageably small dataframe - R does the actual statistics - Close the connection when you’re done

Going the Other Way

- The sqldf package lets you use SQL commands on dataframes
- Mostly useful if you already know SQL better than R...

Summary

- A database is basically a way of dealing efficiently with lots of potentially huge dataframes
- SQL is the standard language for telling databases what to do, especially what queries to run
- Everything in an SQL query is something we’ve practiced already in R
  - subsetting/selection, aggregation, merging, ordering
- Connect R to the database, send it an SQL query, analyse the returned dataframe