Using Databases in R

Marc Carlson
Fred Hutchinson Cancer Research Center

17-18 February, 2011
Introduction

Basic SQL

Using SQL from within R
Relational Databases

Relational database basics

- Data stored in *tables*
- Tables related through *keys*
- Relational model called a *schema*
- Tables designed to avoid redundancy

Beneficial uses by R packages

- Out-of-memory data storage
- Fast access to data subsets
- Databases accessible by other software
Uses of Relational Databases in Bioconductor

Annotation packages

- Organism, genome (e.g. org.Hs.eg.db)
- Microarray platforms (e.g. hgu95av2.db)
- Homology (e.g. hom.Hs.inp.db)

Software packages

- Transcript annotations (e.g. GenomicFeatures)
- NGS experiments (e.g. Genominator)
- Annotation infrastructure (e.g. AnnotationDbi)
What do I mean by relational?
From R it looks pretty simple

Making a TranscriptDb object

```r
> library(GenomicFeatures)
> mm9KG <-
>     makeTranscriptDbFromUCSC(genome = "mm9",
>                              tablename = "knownGene")
```

Saving and Loading

```r
> saveFeatures(mm9KG, file="mm9KG.sqlite")
> mm9KG <-
>     loadFeatures(system.file("extdata", "mm9KG.sqlite",
>                              package = "AdvancedR2011"))
```
There are even accessors etc.

```r
> head(transcripts(mm9KG), 3)

GRanges with 3 ranges and 2 elementMetadata values

<table>
<thead>
<tr>
<th>seqnames</th>
<th>ranges</th>
<th>strand</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Rle&gt;</td>
<td>&lt;IRanges&gt;</td>
<td>&lt;Rle&gt;</td>
</tr>
<tr>
<td>[1]</td>
<td>chr9 [3215314, 3215339]</td>
<td>+</td>
</tr>
<tr>
<td>[2]</td>
<td>chr9 [3335231, 3385846]</td>
<td>+</td>
</tr>
<tr>
<td>[3]</td>
<td>chr9 [3335473, 3343608]</td>
<td>+</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tx_id</th>
<th>tx_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>24312 uc009oas.1</td>
</tr>
<tr>
<td>[2]</td>
<td>24315 uc009oat.1</td>
</tr>
<tr>
<td>[3]</td>
<td>24313 uc009oau.1</td>
</tr>
</tbody>
</table>

seqlengths

<table>
<thead>
<tr>
<th>chr1 ... chrY_random</th>
</tr>
</thead>
<tbody>
<tr>
<td>197195432 ... 58682461</td>
</tr>
</tbody>
</table>
But what actually happens when we do this?

```r
> options(verbose=TRUE)
> txs <- transcripts(mm9KG)
```

**SQL QUERY:** SELECT `tx_chrom`, `tx_start`, `tx_end`, `tx_strand`, transcript.``tx_id`` AS `tx_id` , `tx_name` FROM transcript ORDER BY `tx_chrom`, `tx_strand`, `tx_start`, `tx_end`

Notice how the database query is pretty simple?

- DB joins promote flexible access
- BUT: there can be a cost if using A LOT of joins
- Therefore (in this case) a hybrid approach: Retrieve relevant records and subset in R
Remember: our DB schema has more than 1 table

<table>
<thead>
<tr>
<th>Table</th>
<th>Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>transcript</td>
<td>tx_id INTEGER PRIMARY KEY, tx_name TEXT NULL, tx_chrom TEXT NOT NULL,</td>
</tr>
<tr>
<td></td>
<td>tx_strand TEXT NOT NULL, tx_start INTEGER NOT NULL, tx_end INTEGER NOT NULL,</td>
</tr>
<tr>
<td></td>
<td>FOREIGN KEY (tx_chrom) REFERENCES chrominfo (chrom)</td>
</tr>
<tr>
<td>chrominfo</td>
<td>chrom_id INTEGER PRIMARY KEY, chrom TEXT UNIQUE NOT NULL, length INTEGER NULL</td>
</tr>
<tr>
<td>gene</td>
<td>gene_id TEXT NOT NULL, tx_id INTEGER NOT NULL,</td>
</tr>
<tr>
<td></td>
<td>UNIQUE (gene_id, tx_id), FOREIGN KEY (tx_id) REFERENCES transcript</td>
</tr>
<tr>
<td>splicing</td>
<td>tx_id INTEGER NOT NULL, exon_rank INTEGER NOT NULL, exon_id INTEGER NOT NULL,</td>
</tr>
<tr>
<td></td>
<td>exon_id INTEGER NOT NULL, cds_id INTEGER NOT NULL,</td>
</tr>
<tr>
<td></td>
<td>UNIQUE (tx_id, exon_rank), FOREIGN KEY (tx_id) REFERENCES transcript</td>
</tr>
<tr>
<td>exon</td>
<td>exon_id INTEGER PRIMARY KEY, exon_name TEXT NULL, exon_chrom TEXT NOT NULL,</td>
</tr>
<tr>
<td></td>
<td>exon_strand TEXT NOT NULL, exon_start INTEGER NOT NULL, exon_end INTEGER NOT NULL,</td>
</tr>
<tr>
<td></td>
<td>FOREIGN KEY (exon_chrom) REFERENCES chrominfo (chrom)</td>
</tr>
</tbody>
</table>
Structured Query Language (SQL) is the most common language for interacting with relational databases.
Database Retrieval

Single table selections

```
SELECT * FROM gene;
SELECT gene_id, gene._tx_id FROM gene;

SELECT * FROM gene WHERE _tx_id=49245;
SELECT * FROM transcript WHERE tx_name LIKE '%oap.1';
```

Inner joins

```
SELECT gene.gene_id,transcript._tx_id
    FROM gene, transcript
    WHERE gene._tx_id=transcript._tx_id;

SELECT g.gene_id,t._tx_id
    FROM gene AS g, transcript AS t
    WHERE g._tx_id=t._tx_id
    AND t._tx_id > 10;
```
Database Modifications

CREATE TABLE

CREATE TABLE foo (  
id INTEGER,  
string TEXT
);

INSERT

INSERT INTO foo (id, string) VALUES (1, "bar");

CREATE INDEX

CREATE INDEX fooInd1 ON foo(id);
The `DBI` package

- Provides a nice generic access to databases in R
- Many of the functions are convenient and simple to use
Some popular DBI functions

```r
> library(RSQLite)  # loads DBI too, (but we need both)
> drv <- SQLite()
> con <- dbConnect(drv, dbname=system.file("extdata", "mm9KG.sqlite", package="AdvancedR2011")
> dbListTables(con)

[1] "cds" "chrominfo"
[3] "exon" "gene"
[5] "metadata" "splicing"
[7] "transcript"

> dbListFields(con,"transcript")

[1] "_tx_id" "tx_name" "tx_chrom"
[4] "tx_strand" "tx_start" "tx_end"
```
The dbGetQuery approach

> dbGetQuery(con, "SELECT * FROM transcript LIMIT 3")

<table>
<thead>
<tr>
<th>_tx_id</th>
<th>tx_name</th>
<th>tx_chrom</th>
<th>tx_strand</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>uc009oap.1</td>
<td>chr9</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>uc009oao.1</td>
<td>chr9</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>uc009oaq.1</td>
<td>chr9</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tx_start</th>
<th>tx_end</th>
</tr>
</thead>
<tbody>
<tr>
<td>3186316</td>
<td>3186344</td>
</tr>
<tr>
<td>3133847</td>
<td>3199799</td>
</tr>
<tr>
<td>3190269</td>
<td>3199799</td>
</tr>
</tbody>
</table>
The dbSendQuery approach

If you use result sets, you also need to put them away

```r
> res <- dbSendQuery(con, "SELECT * FROM transcript")
> fetch(res, n= 3)

   _tx_id tx_name tx_chrom tx_strand
1  24308  uc009oap.1 chr9    -
2  24309  uc009oao.1 chr9    -
3  24310  uc009oaq.1 chr9    -

   tx_start  tx_end
1  3186316 3186344
2  3133847 3199799
3  3190269 3199799

> dbClearResult(res)

[1] TRUE
```

Calling fetch again will get the next three results. This allows for simple iteration.
Setting up a new DB

First, let's close the connection to our other DB:

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

Then let's make a new database. Notice that we specify the database name with "dbname". This allows it to be written to disc instead of just memory.

```r
> drv <- SQLite()
> con <- dbConnect(drv, dbname="myNewDb.sqlite")
```

Once you have this, you may want to make a new table

```r
> dbGetQuery(con, "CREATE Table foo (id INTEGER, string TEXT)"
NULL
```
The *RSQLite* package

- Provides SQLite access for R
- Much better support for complex inserts
Prepared queries

```r
> data <- data.frame(c(226089, 66745),
+       c("C030046E11Rik", "Trpd5213"),
+       stringsAsFactors=FALSE)
> names(data) <- c("id", "string")
> sql <- "INSERT INTO foo VALUES ($id, $string)"
> dbBeginTransaction(con)

[1] TRUE

> dbGetPreparedQuery(con, sql, bind.data = data)

NULL

> dbCommit(con)

[1] TRUE

Notice that we want strings instead of factors in our data.frame
```
in SQLite, you can ATTACH Dbs

The SQL what we want looks quite simple:

```
ATTACH "mm9KG.sqlite" AS db;
```

So we just need to do something like this:

```r
> db <- system.file("extdata", "mm9KG.sqlite",
+ package="AdvancedR2011")
> dbGetQuery(con, sprintf("ATTACH '%s' AS db",db))
NULL
```
You can join across attached Dbs

The SQL this looks like:

```
SELECT * FROM db.gene AS dbg, foo AS f
WHERE dbg.gene_id=f.id;
```

Then in R:

```r
sql <- "SELECT * FROM db.gene AS dbg,
+ foo AS f WHERE dbg.gene_id=f.id"
> res <- dbGetQuery(con, sql)
> res
```

```
  gene_id _tx_id  id    string
  1 226089 48508 226089 C030046E11Rik
  2 226089 48509 226089 C030046E11Rik
  3 226089 48511 226089 C030046E11Rik
  4 226089 48510 226089 C030046E11Rik
  5 66745 48522 66745 Trpd5213
```
The End.
No seriously, that was it.
There is nothing "back there" that you need to see...
That was the end.
Using XML in R: the XML package (using xpath)

```r
## 1st assemble a URL to use NCBI's web services
entrezGenes = c(1,100008564)
library(XML)
xsep <- paste(entrezGenes, collapse=","
url <- paste(baseUrl,"&id=",xsep,"&retmode=xml", sep="")
url


## Then parse this XML for use by xpathApply
EGSet <- xmlParse(url)
## Then you can use xpathApply() to parse using xpath expressions
## xmlValue() returns the contents of that tag
speciesNames <- unlist(xpathApply(EGSet, "//Org-ref_taxname", xmlValue))
speciesNames

[1] "Homo sapiens" "Mus musculus"
```
[1] TRUE