

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

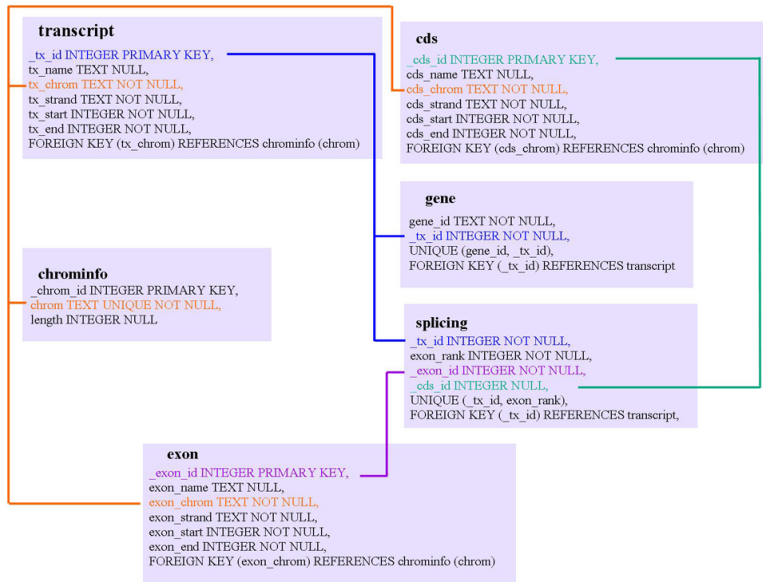
Annotation packages

- ▶ 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?



SQL in 3 slides

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);
```


Some extra tricks

You can use AS to rename parts of a query

```
SELECT * FROM gene AS g;
```

You can use ORDER BY to ensure that things are sorted

```
SELECT * FROM gene ORDER BY gene_id;
```

You can use () to indicate sub-queries. This can be useful when you have more complex queries to express or if you need to combine different kinds of joins into a single query.

```
SELECT * FROM transcript as t, (SELECT * FROM gene) AS g  
WHERE g._tx_id = t._tx_id;
```

It's sometimes a good idea to use a stand-alone client

- ▶ SQLite3 demo

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

```
> 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")
```

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

	<code>tx_start</code>	<code>tx_end</code>
1	3186316	3186344
2	3133847	3199799
3	3190269	3199799

The dbSendQuery approach

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

```
> 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, lets close the connection to our other DB:

```
> dbDisconnect(con)
```

```
[1] TRUE
```

Then lets 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.

```
> drv <- SQLite()
```

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

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

```
> 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

```
> 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 (SQLite specific) SQL that we want looks quite simple:

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

So we just need to do something like this:

```
> 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:

```
> 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.

[1] FALSE