Using Databases in R

Marc Carlson

Fred Hutchinson Cancer Research Center

May 20, 2010
Introduction

Example Databases: The GenomicFeatures Package

Basic SQL

Using SQL from within R
Outline

Introduction

Example Databases: The GenomicFeatures Package

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)
Outline

Introduction

Example Databases: The GenomicFeatures Package

Basic SQL

Using SQL from within R
The GenomicFeatures package

What is does

- Builds databases on the fly as needed.
- Supported sources: UCSC tracks, biomaRt, or custom.
- Has accessor methods to make it easy to use this data.

Why it was built

- Needed a generic way to support access to multiple resources
- Users needed reproducible data for research
- Data needed to be formatted in a convenient way

Why it uses a database

- Annotation data is highly relational
- Other annotation data is also stored in SQLite databases
- Allows access from disc (which can free up memory)
What do I mean by relational?
TranscriptDb Basics

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 = "AdvancedR"))
```
Accessing the TranscriptDb

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

GRanges with 3 ranges and 2 elementMetadata values
tx_id
seqnames ranges strand |     tx_id
    <Rle>   <IRanges>  <Rle> |     <integer>
[1]   chr9  [3215314, 3215339]   + |     24312

  tx_name
<character>
[1]  uc009oas.1
[2]  uc009oat.1
[3]  uc009oau.1

  seqlengths
chr1  chr2 ...   chrX_random  chrY_random
  197195432  181748087 ...       1785075           58682461
```
What actually happens when we do this?

```r
> options(verbos=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 is a cost if using A LOT of joins
- Therefore (in this case) a hybrid approach: Retrieve relevant records and subset in R
Our database schema

**transcript**
- _tx_id INTEGER PRIMARY KEY,
- tx_name TEXT NULL,
- tx_chrom TEXT NOT NULL,
- tx_strand TEXT NOT NULL,
- tx_start INTEGER NOT NULL,
- tx_end INTEGER NOT NULL,
- FOREIGN KEY (tx_chrom) REFERENCES chrominfo (chrom)

**cds**
- _cds_id INTEGER PRIMARY KEY,
- cds_name TEXT NULL,
- cds_chrom TEXT NOT NULL,
- cds_strand TEXT NOT NULL,
- cds_start INTEGER NOT NULL,
- cds_end INTEGER NOT NULL,
- FOREIGN KEY (cds_chrom) REFERENCES chrominfo (chrom)

**chrominfo**
- chrom_id INTEGER PRIMARY KEY,
- chrom TEXT UNIQUE NOT NULL,
- length INTEGER NULL

**gene**
- gene_id TEXT NOT NULL,
- _tx_id INTEGER NOT NULL,
- UNIQUE (gene_id, _tx_id),
- FOREIGN KEY (_tx_id) REFERENCES transcript

**splicing**
- tx_id INTEGER NOT NULL,
- exon_rank INTEGER NOT NULL,
- exon_id INTEGER NOT NULL,
- _cds_id INTEGER NULL,
- UNIQUE (_tx_id, exon_rank),
- FOREIGN KEY (_tx_id) REFERENCES transcript,

**exon**
- exon_id INTEGER PRIMARY KEY,
- exon_name TEXT NULL,
- exon_chrom TEXT NOT NULL,
- exon_strand TEXT NOT NULL,
- exon_start INTEGER NOT NULL,
- exon_end INTEGER NOT NULL,
- FOREIGN KEY (exon_chrom) REFERENCES chrominfo (chrom)
Outline

Introduction

Example Databases: The GenomicFeatures Package

Basic SQL

Using SQL from within R
Structured Query Language (SQL) is the most common language for interacting with relational databases.
Database Retrieval

Single table selections

```sql
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

```sql
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;
```
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);
Outline

Introduction

Example Databases: The GenomicFeatures Package

Basic SQL

Using SQL from within R
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 <- dbDriver("SQLite")
> con <- dbConnect(drv, dbname=system.file("extdata", + "mm9KG.sqlite", package="AdvancedR"))
> dbListTables(con)

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

> dbListFields(con,"transcript")

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

```r
> 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>
<th>tx_start</th>
<th>tx_end</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>24308 uc009oap.1</td>
<td>chr9</td>
<td>-</td>
<td>3186316</td>
<td>3186344</td>
</tr>
<tr>
<td>2</td>
<td>24309 uc009oao.1</td>
<td>chr9</td>
<td>-</td>
<td>3133847</td>
<td>3199799</td>
</tr>
<tr>
<td>3</td>
<td>24310 uc009oaq.1</td>
<td>chr9</td>
<td>-</td>
<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)

<table>
<thead>
<tr>
<th>_tx_id</th>
<th>tx_name</th>
<th>tx_chrom</th>
<th>tx_strand</th>
<th>tx_start</th>
<th>tx_end</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>uc009oap.1</td>
<td>chr9</td>
<td>-</td>
<td>3186316</td>
<td>3186344</td>
</tr>
<tr>
<td>2</td>
<td>uc009oao.1</td>
<td>chr9</td>
<td>-</td>
<td>3133847</td>
<td>3199799</td>
</tr>
<tr>
<td>3</td>
<td>uc009oaq.1</td>
<td>chr9</td>
<td>-</td>
<td>3190269</td>
<td>3199799</td>
</tr>
</tbody>
</table>

> dbClearResult(res)

[1] TRUE
```

Calling `fetch` again will get the next three results. This allows for simple iteration.
Select the exons from the minus strand of chromosome 9.
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 <- dbDriver("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
Create a database and then put a table in it called genePheno to store the genes mutated and a phenotypes associated with each. Plan for genePheno to hold the following gene IDs and phenotypes (as a toy example):

```r
data = data.frame(id=c(69773,20586,258822,18315),
                  string=c("Dead",
                          "Alive",
                          "Dead",
                          "Alive"),
                  stringsAsFactors=FALSE)
```
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
```
Your turn part 3

Now take a moment to insert that data into your database.
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="AdvancedR")
> dbGetQuery(con, sprintf("ATTACH '%s' AS db",db))
NULL
```
You can join across attached Dbs

The SQL this looks like:

```sql
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 Trpd52l3
```
Your turn part 4

Now create a cross join to your database and extract the _tx_id 's from the gene table there using your gene IDs as a foreign key.
Your turn part 5

Now connect your cross join to the transcript table in the database and extract the fields from that table while still using your gene IDs as a foreign key.