

Using Annotations in Bioconductor

Marc Carlson

Fred Hutchinson Cancer Research Center

July 30, 2010

Bioconductor Annotation Packages

AnnotationDbi

AnnotationDbi Basics

Working with GO.db

SQL databases

Basic SQL

Using SQL from within R

Outline

Bioconductor Annotation Packages

AnnotationDbi

AnnotationDbi Basics

Working with GO.db

SQL databases

Basic SQL

Using SQL from within R

Bioconductor annotation packages

Major types of annotation in Bioconductor.

AnnotationDbi packages:

- ▶ Organism level: [org.Mm.eg.db](http://org.mmm.org).
- ▶ Platform level: hgu133plus2.db.
- ▶ System-biology level: [GO.db](http://go.db) or [KEGG.db](http://kegg.db).
- ▶ Transcript centric annotations: [GenomicFeatures](http://genomicfeatures).

biomaRt:

- ▶ Query web-based 'biomart' resource for genes, sequence, SNPs, and etc.

Other packages:

- ▶ rtracklayer – export to UCSC web browsers.

Outline

Bioconductor Annotation Packages

AnnotationDbi

AnnotationDbi Basics

Working with GO.db

SQL databases

Basic SQL

Using SQL from within R

AnnotationDbi

AnnotationDbi is a software package that enables the package annotations:

- ▶ Each supported package contains a database.
- ▶ AnnotationDbi allows access to that data via Bimap objects.
- ▶ Some databases depend on the databases in other packages.

Organism-level annotation

There are a number of organism annotation packages with names starting with `org`, e.g., `org.Hs.eg.db` – genome-wide annotation for human.

```
> library(org.Hs.eg.db)
> org.Hs.eg()
> org.Hs.eg_dbInfo()
> org.Hs.egGENENAME
> org.Hs.eg_dbschema()
```

platform based packages (chip packages)

There are a number of platform or chip specific annotation packages named after their respective platforms, e.g. [hgu95av2.db](#) - annotations for the hgu95av2 Affymetrix platform.

- ▶ These packages appear to contain a lot of data but it's an illusion.

```
> library(hgu95av2.db)
> hgu95av2()
> hgu95av2_dbInfo()
> hgu95av2GENENAME
> hgu95av2_dbschema()
```


Kinds of annotation

What can you hope to extract from an annotation package?

- ▶ GO IDs: [GO](#)
- ▶ KEGG pathway IDs: [PATH](#)
- ▶ Gene Symbols: [SYMBOL](#)
- ▶ Chromosome start and stop locs: [CHRLOC](#) and [CHRLOCEND](#)
- ▶ Alternate Gene Symbols: [ALIAS](#)
- ▶ Associated Pubmed IDs: [PMID](#)
- ▶ RefSeq IDs: [REFSEQ](#)
- ▶ Unigene IDs: [UNIGENE](#)
- ▶ PFAM IDs: [PFAM](#)
- ▶ Prosite IDs: [PROSITE](#)
- ▶ ENSEMBL IDs: [ENSEMBL](#)

Outline

Bioconductor Annotation Packages

AnnotationDbi

AnnotationDbi Basics

Working with GO.db

SQL databases

Basic SQL

Using SQL from within R

Basic Bimap structure and getters

Bimaps create a mapping from one set of keys to another. And they can easily be searched.

- ▶ `toTable`: converts a Bimap to a `data.frame`
- ▶ `get`: pulls data from a Bimap
- ▶ `mget`: pulls data from a Bimap for multiple things at once

```
> head(toTable(hgu95av2SYMBOL))  
> get("38187_at", hgu95av2SYMBOL)  
> mget(c("38912_at", "38187_at"), hgu95av2SYMBOL, ifnotfound=NA)
```

Reversing and subsetting Bimaps

Bimaps can also be reversed and subsetted:

- ▶ `revmap`: reverses a Bimap
- ▶ `[[,[:` Bimaps are subtable.

```
> ##revmap
> mget(c("NAT1", "NAT2"), revmap(hgu95av2SYMBOL), ifnotfound=NA)
> ##subsetting
> head(toTable(hgu95av2SYMBOL[1:3]))
> hgu95av2SYMBOL[["1000_at"]]
> revmap(hgu95av2SYMBOL)[["MAPK3"]]
> ##Or you can combine things
> toTable(hgu95av2SYMBOL[c("38912_at", "38187_at")])
```

Annotation exercise 1

Find the gene symbol, chromosome position and KEGG pathway ID for "1003_s_at".

Annotation exercise 1 solution

```
> library(hgu95av2.db)
> get("1003_s_at",hgu95av2SYMBOL)
> get("1003_s_at",hgu95av2CHRLoc)
> get("1003_s_at",hgu95av2PATH)
> ##OR if you like data frames:
> toTable(hgu95av2SYMBOL["1003_s_at"])
> toTable(hgu95av2CHRLoc["1003_s_at"])
> toTable(hgu95av2PATH["1003_s_at"])
```

Bimap keys

Bimaps create a mapping from one set of keys to another. Some important methods include:

- ▶ `keys`: centralID for the package (directional)
- ▶ `Lkeys`: centralID for the package (probe ID or gene ID)
- ▶ `Rkeys`: centralID for the package (attached data)

```
> keys(hgu95av2SYMBOL[1:4])  
> Lkeys(hgu95av2SYMBOL[1:4])  
> Rkeys(hgu95av2SYMBOL)[1:4]
```


More Bimap structure

Not all keys have a partner (or are mapped)

- ▶ `mappedkeys`: which of the key are mapped (directional)
- ▶ `mappedLkeys mappedRkeys`: which keys are mapped (absolute reference)
- ▶ `count.mappedkeys`: Number of mapped keys (directional)
- ▶ `count.mappedLkeys, count.mappedRkeys`: Number of mapped keys (absolute)

```
> mappedkeys(hgu95av2SYMBOL[1:10])  
> mappedLkeys(hgu95av2SYMBOL[1:10])  
> mappedRkeys(hgu95av2SYMBOL[1:10])  
> count.mappedkeys(hgu95av2SYMBOL[1:100])  
> count.mappedLkeys(hgu95av2SYMBOL[1:100])  
> count.mappedRkeys(hgu95av2SYMBOL[1:100])
```

Bimap Conversions

How to handle conversions from Bimaps to lists

- ▶ `as.list`: converts a Bimap to a `list`
- ▶ `unlist2`: unlists a `list` minus the name-mangling.

```
> as.list(hgu95av2SYMBOL[c("38912_at", "38187_at")])
> unlist(as.list(hgu95av2SYMBOL[c("38912_at", "38187_at")]))
> unlist2(as.list(hgu95av2SYMBOL[c("38912_at", "38187_at")]))
> ##but what happens when there are
> ##repeating values for the left key?
> unlist(as.list(revmap(hgu95av2SYMBOL)[c("STAT1", "PTGER3")]))
> ##unlist2 can help with this
> unlist2(as.list(revmap(hgu95av2SYMBOL)[c("STAT1", "PTGER3")]))
```

Annotation exercise 2

Gene symbols are often recycled by other genes making them a poor choice for identifiers. Using what you have learned, the SYMBOL Bimap, along with the `lapply`, `length` and `sort` functions, determine which gene symbols in `hgu95av2` are the worst offenders.

Annotation exercise 2 solution

```
> badRank <- lapply(as.list(revmap(hgu95av2SYMBOL))), length)
> tail(sort(unlist(badRank)))
```

toggleProbes

How to hide/unhide ambiguous probes.

- ▶ `toggleProbes`: hides or displays the probes that have multiple mappings to genes.

```
> ## How many probes?  
> dim(hgu95av2ENTREZID)  
> ## Make a mapping with multiple probes exposed  
> multi <- toggleProbes(hgu95av2ENTREZID, "all")  
> ## How many probes?  
> dim(multi)  
> ## Make a mapping with ONLY multiple probes exposed  
> multiOnly <- toggleProbes(multi, "multiple")  
> ## How many probes?  
> dim(multiOnly)  
> ## Then make a mapping with ONLY single mapping probes  
> singleOnly <- toggleProbes(multiOnly, "single")  
> ## How many probes?  
> dim(singleOnly)
```

Annotation exercise 3

Using the knowledge that Entrez IDs are good IDs that can be used to define genes uniquely, find the probe that maps to the largest number of different genes on hgu95av2.

Annotation exercise 3 solution

```
> mult <- toggleProbes(hgu95av2ENTREZID, "multi")  
> dim(mult)  
> multRank <- lapply(as.list(mult), length)  
> tail(sort(unlist(multRank)))
```

Outline

Bioconductor Annotation Packages

AnnotationDbi

AnnotationDbi Basics

Working with GO.db

SQL databases

Basic SQL

Using SQL from within R

GO

Some important considerations about the Gene Ontology

- ▶ GO is actually 3 ontologies (CC, BP and MF)
- ▶ Each ontology is a directed acyclic graph.
- ▶ The structure of GO is maintained separately from the genes that these GO IDs are usually used to annotate.

GO to gene mappings are stored in other packages

Mapping Entrez IDs to GO

- ▶ Each ENTREZ ID is associated with up to three GO categories.
- ▶ The objects returned from an ordinary GO mapping are complex.

```
> go <- org.Hs.egGO[["1000"]]
> length(go)
> go[[2]]$GOID
> go[[2]]$Ontology
```

Annotation exercise 4

Use what you have learned to write a function that gets the GOIDs for a particular entrez gene ID, and then returns only their GOID as a named vector. Use `lapply` and `(names)`.

Annotation exercise 4 solution

```
> ##get GOIDs from Hs package.  
> getGOIDs <- function(ids){  
+   require(org.Hs.eg.db)  
+   GOs = mget(ids, org.Hs.egGO, ifnotfound=NA)  
+   unlist2(lapply(GOs,names))  
+ }  
> ##usage example:  
> getGOIDs(c("1","10"))
```

Working with GO.db

- ▶ Encodes the hierarchical structure of GO terms.
- ▶ The mapping between GO terms and individual genes is maintained in the GO mappings from the other packages.
- ▶ the difference between children and offspring is how many generations are represented. Children only nets you one step down the graph.

```
> library(GO.db)
> ls("package:GO.db")
> ## find children
> as.list(GOMFCHILDREN["GO:0008094"])
> ## all the descendants (children, grandchildren, and so on)
> as.list(GOMFOFFSPRING["GO:0008094"])
```


GO helper methods

Using the GO helper methods

- ▶ The GO terms are described in detail in the [GOTERM](#) mapping.
- ▶ The objects returned by GO.db are GOTerms objects, which can make use of helper methods like `GOID`, `Term`, `Ontology` and `Definition` to retrieve various details.
- ▶ You can also pass GOIDs to these helper methods.

```
> ##Mapping a GOTerms object
> go <- GOTERM[1]
> GOID(go)
> Term(go)
> ##OR you can supply GO IDs
> id = c("GO:0007155","GO:0007156")
> GOID(id)
> Term(id)
> Ontology(id)
> Definition(id)
```

Annotation exercise 5

Use what they have learned to write a function that calls your previous function so that it can get GOIDs and then returns the GO definitions.

Annotation exercise 5 solution

```
> ##get GOIDs from Hs package.  
> library(GO.db)  
> getGODefs <- function(ids){  
+   GOids <- getGOIDs(ids)  
+   defs <- Definition(GOids)  
+   names(defs) <- names(GOids)  
+   defs  
+ }  
> ##usage example:  
> getGODefs(c("1", "10"))
```

Outline

Bioconductor Annotation Packages

AnnotationDbi

AnnotationDbi Basics

Working with GO.db

SQL databases

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

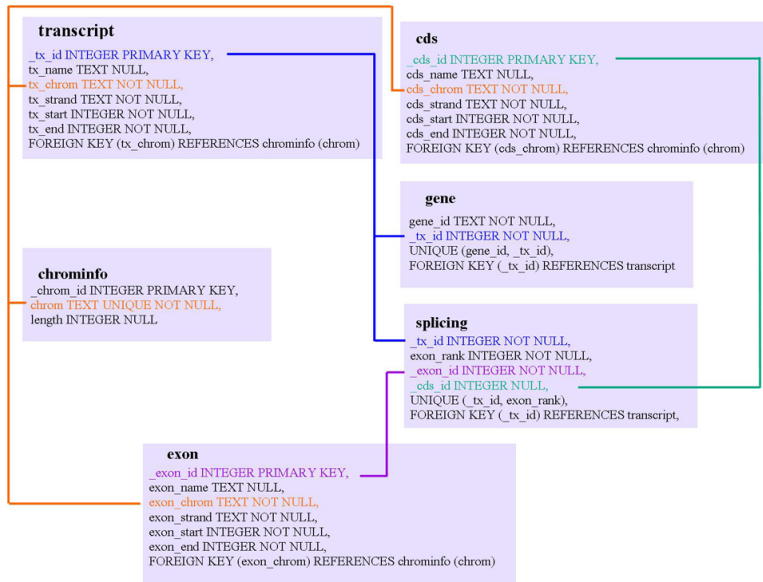
Annotation packages

- ▶ 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`)

A example database schema



Outline

Bioconductor Annotation Packages

AnnotationDbi

AnnotationDbi Basics

Working with GO.db

SQL databases

Basic SQL

Using SQL from within R

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);
```

Outline

Bioconductor Annotation Packages

AnnotationDbi

AnnotationDbi Basics

Working with GO.db

SQL databases

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

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

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

	<code>_tx_id</code>	<code>tx_name</code>	<code>tx_chrom</code>	<code>tx_strand</code>	<code>tx_start</code>	<code>tx_end</code>
1	24308	uc009oap.1	chr9	-	3186316	3186344
2	24309	uc009oao.1	chr9	-	3133847	3199799
3	24310	uc009oaq.1	chr9	-	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	tx_start	tx_end
1	24308	uc009oap.1	chr9	-	3186316	3186344
2	24309	uc009oao.1	chr9	-	3133847	3199799
3	24310	uc009oaq.1	chr9	-	3190269	3199799

```
> dbClearResult(res)
```

```
[1] TRUE
```

Calling fetch again will get the next three results. This allows for simple iteration.

Exercise 6

Connect to the database in these slides and use the schema diagrammed to select the exons from the minus strand of chromosome 9.

Annotation exercise 6 solution

```
> library(RSQLite)
> drv <- dbDriver("SQLite")
> con <- dbConnect(drv, dbname=system.file("extdata",
+                                           "mm9KG.sqlite", package="HTSandGeneCentricLab")
> sql <- "SELECT * FROM exon WHERE exon.exon_strand='- '
+        AND exon.exon_chrom='chr9'"
> res <- dbGetQuery(con, sql)
```

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 <- dbDriver("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
```

Exercise 7

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):

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


Annotation exercise 7 solution

```
> drv <- dbDriver("SQLite")
> dbcon <- dbConnect(drv, dbname="myExDb.sqlite")
> cval <- dbGetQuery(dbcon,
+                     "CREATE Table genePheno
+                     (id INTEGER, string TEXT)")
```


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

Exercise 8

Now take a moment to insert that data into your database.

Annotation exercise 8 solution

```
> data <- data.frame(id=c(69773,20586,258822,18315),  
+                    string=c("Dead",  
+                             "Alive",  
+                             "Dead",  
+                             "Alive"),  
+                    stringsAsFactors=FALSE)  
> sql <- "INSERT INTO genePheno VALUES ($id,$string)"  
> bval <- dbBeginTransaction(dbcon)  
> gval <- dbGetPreparedQuery(dbcon, sql, bind.data = data)  
> cval <- dbCommit(dbcon)
```

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:

```
> db <- system.file("extdata", "mm9KG.sqlite",  
+                   package="HTSandGeneCentricLabs")  
> 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

Exercise 9

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.

Annotation exercise 9 solution

```
> #1st attach
> db <- system.file("extdata", "mm9KG.sqlite",
+                   package="HTSandGeneCentricLabs")
> att <- dbGetQuery(dbcon, sprintf("ATTACH '%s' AS db",db))
> #then select
> sql <- "SELECT * FROM db.gene AS dbg,
+        genePheno AS gp WHERE dbg.gene_id=gp.id"
> res <- dbGetQuery(dbcon, sql)
```

Exercise 10

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.

Annotation exercise 10 solution

```
> sql <- "SELECT * FROM db.gene AS dbg,  
+         genePheno AS gp, transcript AS t  
+         WHERE dbg.gene_id=gp.id  
+         AND dbg._tx_id=t._tx_id"  
> res <- dbGetQuery(dbcon, sql)
```