

Introduction To Bioconductor Annotation Packages

Marc Carlson

September 26, 2012

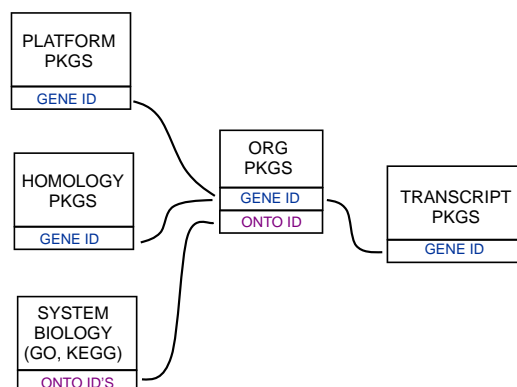


Figure 1: Annotation Packages: the big picture

Bioconductor provides extensive annotation resources. These can be *gene centric*, or *genome centric*. Annotations can be provided in packages curated by *Bioconductor*, or obtained from web-based resources. This vignette is primarily concerned with describing the annotation resources that are available as packages. This includes both how to extract data from them and also what steps are required to expose other databases in a similar fashion.

Gene centric *AnnotationDbi* packages include:

- Organism level: e.g. *org.Mm.eg.db*.
- Platform level: e.g. *hgu133plus2.db*, *hgu133plus2.probes*, *hgu133plus2.cdf*.
- Homology level: e.g. *hom.Dm.inp.db*.

- System-biology level: *GO.db* or *KEGG.db*.

Genome centric *GenomicFeatures* packages include

- Transcriptome level: e.g. *TxDb.Hsapiens.UCSC.hg19.knownGene*
- Generic genome features: Can generate via *GenomicFeatures*

One web-based resource accesses [biomart](#), via the *biomaRt* package:

- Query web-based ‘biomart’ resource for genes, sequence, SNPs, and etc.

The most popular annotation packages have been modified so that they can make use of a new set of methods to more easily access their contents. These four methods are named: `cols`, `keytypes`, `keys` and `select`. And they are described in this vignette. They can currently be used with all chip, organism, and *TranscriptDb* packages along with the popular *GO.db* package.

For the older less popular packages, there are still convenient ways to retrieve the data. The *How to use bimap from the ".db" annotation packages* vignette in the *AnnotationDbi* package is a key reference for learning about how to use bimap objects.

Finally, all of the ‘.db’ (and most other *Bioconductor* annotation packages) are updated every 6 months corresponding to each release of *Bioconductor*. Exceptions are made for packages where the actual resources that the packages are based on have not themselves been updated.

0.1 Organism level packages

An organism level package (an ‘org’ package) uses a central gene identifier (e.g. Entrez Gene id) and contains mappings between this identifier and other kinds of identifiers (e.g. GenBank or Uniprot accession number, RefSeq id, etc.). The name of an org package is always of the form *org.<Ab>.<id>.db* (e.g. *org.Sc.sgd.db*) where *<Ab>* is a 2-letter abbreviation of the organism (e.g. *Sc* for *Saccharomyces cerevisiae*) and *<id>* is an abbreviation (in lower-case) describing the type of central identifier (e.g. *sgd* for gene identifiers assigned by the *Saccharomyces* Genome Database, or *eg* for Entrez Gene ids).

0.2 AnnotationDb objects and the select method

As previously mentioned, a new set of methods have been added that allow a simpler way of extracting identifier based annotations. All the annotation packages that support these new methods expose an object named exactly the same way as the package itself. These objects are collectively called *AnntoationDb* objects, with more specific classes with names such as *OrgDb*, *ChipDb* or *TranscriptDb* objects. The methods that can be applied to these objects are `cols`, `keys`, `keytypes` and `select`.

Exercise 1

Display the *OrgDb* object for the [org.Hs.eg.db](#) package.

Use the `cols` method to discover which sorts of annotations can be extracted from it. Is this the same as the result from the `keytypes` method? Use the `keytypes` method to find out.

Use the `keys` method to extract UNIPROT identifiers and then pass those keys in to the `select` method in such a way that you extract the gene symbol and KEGG pathway information for each.

Solution:

```
R> library(org.Hs.eg.db)
```

```
R> cols(org.Hs.eg.db)
```

```
[1] "ENTREZID"      "ACCNUM"        "ALIAS"         "CHR"
[5] "ENZYME"        "GENENAME"      "MAP"           "OMIM"
[9] "PATH"          "PMID"          "REFSEQ"        "SYMBOL"
[13] "UNIGENE"       "CHRLOC"        "CHRLOCEND"     "PFAM"
[17] "PROSITE"       "ENSEMBL"       "ENSEMBLPROT"   "ENSEMBLTRANS"
[21] "UNIPROT"       "UCSCKG"        "GO"
```

```
R> keytypes(org.Hs.eg.db)
```

```
[1] "ENTREZID"      "ACCNUM"        "ALIAS"         "CHR"
[5] "ENZYME"        "MAP"           "OMIM"          "PATH"
[9] "PMID"          "REFSEQ"        "SYMBOL"        "UNIGENE"
[13] "ENSEMBL"       "ENSEMBLPROT"   "ENSEMBLTRANS" "UNIPROT"
[17] "UCSCKG"        "GO"
```

```
R> uniKeys <- head(keys(org.Hs.eg.db, keytype="UNIPROT"))
```

```
R> cols <- c("SYMBOL", "PATH")
```

```
R> select(org.Hs.eg.db, keys=uniKeys, cols=cols, keytype="UNIPROT")
```

	UNIPROT	SYMBOL	PATH
1	P04217	A1BG	<NA>
2	P01023	A2M	04610
3	F5H5R8	NAT1	01100
4	F5H5R8	NAT1	00232
5	F5H5R8	NAT1	00983
6	P18440	NAT1	01100
7	P18440	NAT1	00232
8	P18440	NAT1	00983
9	Q400J6	NAT1	01100
10	Q400J6	NAT1	00232
11	Q400J6	NAT1	00983
12	A4Z6T7	NAT2	00232
13	A4Z6T7	NAT2	00983
14	A4Z6T7	NAT2	01100

R>

0.3 TranscriptDb packages

A *TranscriptDb* package (a 'TxDb' package) connects a set of genomic coordinates to various transcript oriented features. The package can also contain Identifiers to features such as genes and transcripts, and the internal schema describes the relationships between these different elements. All TranscriptDb containing packages follow a specific naming scheme that tells where the data came from as well as which build of the genome it comes from.

Exercise 2

Display the *TranscriptDb* object for the *TxDb.Hsapiens.UCSC.hg19.knownGene* package.

As before, use the *cols* and *keytypes* methods to discover which sorts of annotations can be extracted from it.

Use the *keys* method to extract just a few gene identifiers and then pass those keys in to the *select* method in such a way that you extract the transcript ids and transcript starts for each.

Solution:

```
R> library(TxDb.Hsapiens.UCSC.hg19.knownGene)
R> txdb <- TxDb.Hsapiens.UCSC.hg19.knownGene
R> txdb
```

```
TranscriptDb object:
| Db type: TranscriptDb
| Supporting package: GenomicFeatures
| Data source: UCSC
| Genome: hg19
| Genus and Species: Homo sapiens
| UCSC Table: knownGene
| Resource URL: http://genome.ucsc.edu/
| Type of Gene ID: Entrez Gene ID
| Full dataset: yes
| miRBase build ID: GRCh37
| transcript_nrow: 80922
| exon_nrow: 286852
| cds_nrow: 235842
| Db created by: GenomicFeatures package from Bioconductor
| Creation time: 2012-03-12 21:45:23 -0700 (Mon, 12 Mar 2012)
| GenomicFeatures version at creation time: 1.7.30
| RSQLite version at creation time: 0.11.1
| DBSCHEMAVERSION: 1.0
```

```
R> cols(txdb)
```

```
[1] "CDSID"      "CDSNAME"    "CDSCHROM"   "CDSSTRAND"  "CDSSTART"
[6] "CSEND"      "EXONID"     "EXONNAME"   "EXONCHROM"  "EXONSTRAND"
[11] "EXONSTART"  "EXONEND"    "GENEID"     "TXID"       "EXONRANK"
[16] "TXNAME"     "TXCHROM"    "TXSTRAND"   "TXSTART"    "TXEND"
```

```
R> keytypes(txdb)
```

```
[1] "GENEID"    "TXID"      "TXNAME"    "EXONID"    "EXONNAME"  "CDSID"
[7] "CDSNAME"
```

```
R> keys <- head(keys(txdb, keytype="GENEID"))
```

```
R> cols <- c("TXID", "TXSTART")
```

```
R> select(txdb, keys=keys, cols=cols, keytype="GENEID")
```

```
      GENEID  TXID  TXSTART
1          1  72180  58858172
```

```

2          1 72182  58859832
3         10 31373  18248755
4        100 73453  43248163
5       1000 66871  25530930
6       1000 66872  25530930
7      10000  7620 243651535
8      10000  7621 243663021
9      10000  7622 243663021
10 100008586 37647 49217763

```

```
R>
```

As is widely known, in addition to providing access via the `select` method, *TranscriptDb* objects also provide access via the more familiar `transcripts`, `exons`, `cds`, `transcriptsBy`, `exonsBy` and `cdsBy` methods. For those who do not yet know about these other methods, more can be learned by seeing the vignette called: *Making and Utilizing TranscriptDb Objects* in the *GenomicFeatures* package.

0.4 Advanced topic: Creating other kinds of Annotation packages

A few options already exist for generating various kinds of annotation packages. For users who seek to make custom chip packages, users should see the *SQLForge: An easy way to create a new annotation package with a standard database schema.* in the *AnnotationDbi* package. And, for users who seek to make a probe package, there is another vignette called *Creating probe packages* that is also in the *AnnotationDbi* package. And finally, for custom organism packages users should look at the manual page for `makeOrgPackageFromNCBI`. This function will attempt to make you an simplified organism package from NCBI resources. However, this function is not meant as a way to refresh annotation packages between releases. It is only meant for people who are working on less popular model organisms (so that annotations can be made available in this format).

But what if you had another kind of database resource and you wanted to expose it to the world using something like this new `select` method interface? How could you go about this?

The 1st step would be to make a package that contains a SQLite database. For the sake of expediency, let's look at an existing example of this in the *hom.Hs.inp.db* package. If you download this tarball from the website you

can see that it contains a .sqlite database inside of the inst/extdata directory. There are a couple of important details though about this database. The 1st is that we recommend that the database have the same name as the package, but end with the extension .sqlite. The second detail is that we recommend that the metadata table contain some important fields. This is the metadata from the current *hom.Hs.inp.db* package.

	name		value
1	INPSOURCEDATE		29-Apr-2009
2	INPSOURCENAME		Inparanoid Orthologs
3	INPSOURCEURL		http://inparanoid.sbc.su.se/download/current/sqltables/
4	DBSCHEMA		INPARANOID_DB
5	ORGANISM		Homo sapiens
6	SPECIES		Human
7	package		AnnotationDbi
8	Db type		InparanoidDb
9	DBSCHEMAVERSION		2.1

As you can see there are a number of very useful fields stored in the metadata table and if you list the equivalent table for other packages you will find even more useful information than you find here. But the most important fields here are actually the ones called "package" and "Db type". Those fields specify both the name of the package with the expected class definition, and also the name of the object that this database is expected to be represented by in the R session respectively. If you fail to include this information in your metadata table, then `loadDb` will not know what to do with the database when it is called. In this case, the class definition has been stored in the *AnnotationDbi* package, but it could live anywhere you need it too. By specifying the metadata field, you enable `loadDb` to find it.

Once you have set up the metadata you will need to create a class for your package that extends the *AnnotationDb* class. In the case of the

hom.Hs.inp.db package, the class is defined to be a *InparanoidDb* class. This code is inside of *AnnotationDbi*.

```
R> .InparanoidDb <-  
    setRefClass("InparanoidDb", contains="AnnotationDb")
```

Finally the `.onLoad` call for your package will have to contain code that will call the `loadDb` method. This is what it currently looks like in the `Rpackagehom.Hs.inp.db` package.

```
R> sPkgname <- sub(".db$", "", pkgname)  
R> txdb <- loadDb(system.file("extdata", paste(sPkgname,  
    ".sqlite", sep=""), package=pkgname, lib.loc=libname),  
    packageName=pkgname)  
R> dbNewname <- AnnotationDbi::dbObjectName(pkgname, "InparanoidDb")  
R> ns <- asNamespace(pkgname)  
R> assign(dbNewname, txdb, envir=ns)  
R> namespaceExport(ns, dbNewname)
```

When this code is run, the name of the package is used to derive the name for the object. Then that name, is used by `onload` to create an *InparanoidDb* object. This object is then assigned to the namespace for this package so that at load time it will be loaded for the user.

0.5 Creating package accessors

At this point, all that remains is to create the means for accessing the data in the database. This should prove a lot less difficult than it may initially sound. For the new interface, only the four methods that were described earlier are really required: `cols`, `keytypes`, `keys` and `select`.

In order to do this you need to know a small amount of SQL and a few tricks for accessing the database from R. The point of providing these 4 accessors is to give users of these packages a more unified experience when retrieving data from the database. But other kinds of accessors (such as those provided for the *TranscriptDb* objects) may also be warranted.

0.5.1 Getting a connection

If all you know is the name of the SQLite database, then to get a DB connection you need to do something like this:


```
R> drv <- SQLite()
R> library("org.Hs.eg.db")
R> con <- dbConnect(drv, dbname=system.file("extdata", "org.Hs.eg.sqlite",
                                             package = "org.Hs.eg.db"))
R> con
```

But in our case the connection is already here as part of the object:

```
R> str(hom.Hs.inp.db)
```

```
Reference class 'InparanoidDb' [package "AnnotationDbi"] with 2 fields
 $ conn          :Formal class 'SQLiteConnection' [package "RSQLite"] with 1 slots
 .. ..@ Id:<externalptr>
 $ packageName: chr "hom.Hs.inp.db"
and 11 methods,
```

So we can do something like below:

```
R> hom.Hs.inp.db$conn

<SQLiteConnection: DBI CON (3744, 13)>

R> ## or better we can use a helper function to wrap this
R> AnnotationDbi::dbConn(hom.Hs.inp.db)

<SQLiteConnection: DBI CON (3744, 13)>
```

0.5.2 Getting data out

Now we just need to get our data out of the DB. There are several useful functions for doing this. Most of these come from the RSQLite or DBI packages. For the sake of simplicity, I will only discuss those that are immediately useful for exploring and extracting data from a database in this vignette. One pair of useful methods are the `dbListTables` and `dbListFields` which are useful for exploring the schema of a database.

```
R> con <- AnnotationDbi::dbConn(hom.Hs.inp.db)
R> head(dbListTables(con))

[1] "Acyrtosiphon_pisum"    "Aedes_aegypti"
[3] "Anopheles_gambiae"    "Apis_mellifera"
[5] "Arabidopsis_thaliana" "Aspergillus_fumigatus"
```

```
R> dbListFields(con, "Mus_musculus")
```

```
[1] "inp_id"      "clust_id"    "species"     "score"
[5] "seed_status"
```

And for actually executing SQL to retrieve data, you probably want to use something like `dbGetQuery`. The only caveat is that this will actually require you to know a little SQL.

```
R> dbGetQuery(con, "SELECT * FROM metadata")
```

	name	value
1	INPSOURCEDATE	29-Apr-2009
2	INPSOURCENAME	Inparanoid Orthologs
3	INPSOURCEURL	http://inparanoid.sbc.su.se/download/current/sqltables/
4	DBSCHEMA	INPARANOID_DB
5	ORGANISM	Homo sapiens
6	SPECIES	Human
7	package	AnnotationDbi
8	Db type	InparanoidDb
9	DBSCHEMAVERSION	2.1

0.5.3 Some basic SQL

The good news is that SQL is pretty easy to learn. Especially if you are primarily interested in just retrieving data from an existing database. Here is a quick run-down to get you started on writing simple `SELECT` statements. Consider a table that looks like this:

	foo	bar
Table sna:	1	baz
	2	boo

This statement:

```
SELECT bar FROM sna;
```

Tells SQL to get the "bar" field from the "foo" table. If we wanted the other field called "sna" in addition to "bar", we could have written it like this:

```
SELECT foo, bar FROM sna;
```

Or even this (* is a wildcard character here)

```
SELECT * FROM sna;
```

Now lets suppose that we wanted to filter the results. We could also have said something like this:

```
SELECT * FROM sna where bar='boo';
```

That query will only retrieve records from foo that match the criteria for bar. But there are two other things to notice. First notice that a single = was used for testing equality. Second notice that I used single quotes to demarcate the string. I could have also used double quotes, but when working in R this will prove to be less convenient as the whole SQL statement itself will frequently have to be wrapped as a string.

What if we wanted to be more general? Then you can use LIKE. Like this:

```
SELECT * FROM sna where bar LIKE 'boo%';
```

That query will only return records where bar starts with "boo", (the % character is acting as another kind of wildcard in this context)

You will often find that you need to get things from two or more different tables at once. Or, you may even find that you need to combine the results from two different queries. Sometimes these two queries may even come from the same table. In any of these cases, you want to do a join. The simplest and most common kind of join is an inner join. Lets suppose that we have two tables:

	foo	bar
Table sna:	1	baz
	2	boo

	foo	bo
Table fu:	1	hi
	2	ca

And we want to join them where the records match in their corresponding "foo" columns. We can do this query to join them:

```
SELECT * FROM sna,fu WHERE sna.foo=fu.foo;
```

Something else we can do is tidy this up by using aliases like so:

```
SELECT * FROM sna AS s,fu AS f WHERE s.foo=f.foo;
```

This last trick is not very useful in this particular example since the query ended up being longer than we started with, but is still great for other cases where queries can become really long.

0.5.4 Exploring the SQLite database from R

Now that we know both some SQL and also about some of the methods in *DBI* and *RSQLite* we can begin to explore the underlying database from R. How should we go about this? Well the 1st thing we always want to know are what tables are present. We already know how to learn this:

```
R> con <- AnnotationDbi::dbConn(hom.Hs.inp.db)
R> head(dbListTables(con))

[1] "Acyrtosiphon_pisum"    "Aedes_aegypti"
[3] "Anopheles_gambiae"    "Apis_mellifera"
[5] "Arabidopsis_thaliana" "Aspergillus_fumigatus"
```

And we also know that once we have a table we are curious about, we can then look up it's fields using `dbListFields`

```
R> dbListFields(con, "Apis_mellifera")

[1] "inp_id"      "clust_id"    "species"     "score"
[5] "seed_status"
```

And once we know something about which fields are present in a table, we can compose a SQL query. perhaps the most straightforward query is just to get all the results from a given table. We know that the SQL for that should look like:

```
SELECT * FROM Apis_mellifera;
```

So we can now call a query like that from R by using `dbGetQuery`:

```
R> head(dbGetQuery(con, "SELECT * FROM Apis_mellifera"))
```

	inp_id	clust_id	species	score	seed_status
1	XP_623957.2	1	APIME	1	100%
2	ENSP00000262442	1	HOMSA	1	99%
3	ENSP00000300671	1	HOMSA	0.095	
4	XP_001121322.1	2	APIME	1	100%
5	ENSP00000265104	2	HOMSA	1	100%
6	ENSP00000333363	2	HOMSA	0.236	

Exercise 3

Now use what you have learned to explore the *hom.Hs.inp.db* database. The formal scientific name for one of the mosquitos that carry the malaria parasite is *Anopheles gambiae*. Now find the table for that organism in the *hom.Hs.inp.db* database and extract it into R. How many species are present in this table? Inparanoid uses a five letter designation for each species that is composed of the 1st 2 letters of the genus followed by the 1st 3 letters of the species. Using this fact, write a SQL query that will retrieve only records from this table that are from humans (*Homo sapiens*).

Solution:

```
R> head(dbGetQuery(con, "SELECT * FROM Anopheles_gambiae"))
R> ## Then only retrieve human records
R> ## Query: SELECT * FROM Anopheles_gambiae WHERE species='HOMSA'
R> head(dbGetQuery(con, "SELECT * FROM Anopheles_gambiae WHERE species='HOMSA'))
```

0.5.5 Example: creating a cols method

Now lets suppose that we want to define a `cols` method for our *hom.Hs.inp.db* object. And lets also suppose that we want is for it to tell us about the actual organisms for which we can extract identifiers. How could we do that?

```

R> .cols <- function(x){
  con <- AnnotationDbi::dbConn(x)
  list <- dbListTables(con)
  ## drop unwanted tables
  unwanted <- c("map_counts", "map_metadata", "metadata")
  list <- list[!list %in% unwanted]
  ## Then just to format things in the usual way
  toupper(list)
}
R> ## Then make this into a method
R> setMethod("cols", "InparanoidDb", .cols(x))
R> ## Then we can call it
R> cols(hom.Hs.inp.db)

```

Notice how I formatted the output to all uppercase characters? This is just done to make the interface look consistent with what has been done before for the other `select` interfaces. But doing this means that we will have to do a tiny bit of extra work when we implement our other methods.

Exercise 4

Now use what you have learned to try and define a method for `keytypes` on `hom.Hs.inp.db`. The `keytypes` method should return the same results as `cols` (in this case). What if you needed to translate back to the lowercase table names? Also write a quick helper function to do that.

Solution:

```

R> setMethod("keytypes", "InparanoidDb", .cols(x))
R> ## Then we can call it
R> keytypes(hom.Hs.inp.db)
R> ## refactor of .cols
R> .getLCcolnames <- function(x){
  con <- AnnotationDbi::dbConn(x)
  list <- dbListTables(con)
  ## drop unwanted tables
  unwanted <- c("map_counts", "map_metadata", "metadata")
  list <- list[!list %in% unwanted]
}
R> .cols <- function(x){
  list <- .getLCcolnames(x)
  ## Then just to format things in the usual way

```

```

    toupper(list)
  }
R> ## Test:
R> cols(hom.Hs.inp.db)
R> ## new helper function:
R> .getTableNames <- function(x){
  LC <- .getLCcolnames(x)
  UC <- .cols(x)
  names(UC) <- LC
  UC
}
R> .getTableNames(hom.Hs.inp.db)

```

Exercise 5

Now define a method for *keys* on *hom.Hs.inp.db*. The *keys* method should return the keys from a given organism based on the appropriate keytype. Since each table has rows that correspond to both human and non-human IDs, it will be necessary to filter out the human rows from the result

Solution:

```

R> .keys <- function(x, keytype){
  ## translate keytype back to table name
  tabNames <- .getTableNames(x)
  lckeytype <- names(tabNames[tabNames %in% keytype])
  ## get a connection
  con <- AnnotationDbi::dbConn(x)
  sql <- paste("SELECT inp_id FROM", lckeytype, "WHERE species!='HOMSA'")
  res <- dbGetQuery(con, sql)
  as.vector(t(res))
}
R> setMethod("keys", "InparanoidDb", .keys(x, keytype))
R> ## Then we can call it
R> keys(hom.Hs.inp.db, "TRICHOPLAX_ADHAERENS")

```