

AnnotationDbi

Herve Pages, Marc Carlson, Seth Falcon, Nianhua Li

June 15, 2009

1 Introduction

1.0.1 General remarks

AnnotationDbi is used primarily to create maps that allow easy access from R to underlying annotation databases. AnnotationDbi introduces a new future for the Bioconductor annotation data packages by changing the paradigm that is used for exchanging annotations.

The largest difference between the older style of annotation packages and the newer ones is the decision to place a real database inside of each package. This is an improved design, because ultimately, the amount of annotation as well as the complexity of this information is likely to increase with time. And perhaps more importantly, this large amount of information needs to be organized in a flexible way in order to maximise its usefulness in a wide array of different circumstances. Since databases were created to solve problems just like this, the benefits of using real databases as the ultimate data structures for annotation packages is self evident.

For this remake of these classic annotation packages, the decision has been made to keep these databases gene centric rather than transcript centric or protein centric.

1.0.2 Database Schemas

For developers, a lot of the benefits of having the information loaded into a real database will require some knowledge about the database schema. For this reason the schemas that were used in the creation of each database type are included in AnnotationDbi. The currently supported schemas are listed in the DBschemas directory of AnnotationDbi. But it is also possible to simply print out the schema that a give package is currently using by using its "_dbschema" method.

Please note that there is one schema for each kind of package. These schemas specify which tables and indices will be present for each package of that type. The schema that a particular package is using is also listed when you type the name of the package as a function to obtain quality control information.

The code to make most kinds of the new database packages is also included in `AnnotationDbi`. Please see the vignette on `SQLForge` for more details on how to make additional database packages.

1.0.3 Internal Design

The current design of these packages is as deliberately simple as it is gene centric. Each table in the database contains a unique kind of information and also an internal identifier called `_id`. The internal `_id` has no meaning outside of the context of a single database. But `_id` does connect all the data within a single database.

As an example if we wanted to connect the values in the `genes` table with the values in the `kegg` table, we could simply join the two tables using the internal `_id` column. It is very important to note however that `_id` does not have any absolute significance. That is, it has no meaning outside of the context of the database where it is used. It is tempting to think that an `_id` could have such significance because within a single database, it looks and behaves similarly to an `entrez` gene ID. But `_id` is definitely NOT an `entrez` gene ID. The `entrez` gene IDs are in another table entirely, and can be connected to using the internal `_id` just like all the other meaningful information inside these databases.

2 Examples

2.0.4 Basic information

The `AnnotationDbi` package provides an interface to SQLite-based annotation packages. Each SQLite-based annotation package (identified by a “.db” suffix in the package name) contains a number of `AnnDbBimap` objects in place of the `environment` objects found in the old-style environment-based annotation packages. The API provided by `AnnotationDbi` allows you to treat the `AnnDbBimap` objects like `environment` instances. For example, the functions `[`, `get`, `mget`, and `ls` all behave the same as with the old-style packages. In addition, new methods like `[, toTable, subset` and others provide some additional flexibility in accessing the annotation data.

```
R> library("hgu95av2.db")
```

The same basic set of objects is provided with the db packages:

```
R> ls("package:hgu95av2.db")
```

```
[1] "hgu95av2"                "hgu95av2_dbconn"
[3] "hgu95av2_dbfile"        "hgu95av2_dbInfo"
[5] "hgu95av2_dbschema"      "hgu95av2ACCNUM"
[7] "hgu95av2ALIAS2PROBE"    "hgu95av2CHR"
[9] "hgu95av2CHRLNGTHS"      "hgu95av2CHRLC"
[11] "hgu95av2CHRLCEND"       "hgu95av2ENSEMBL"
[13] "hgu95av2ENSEMBL2PROBE"  "hgu95av2ENTREZID"
[15] "hgu95av2ENZYME"         "hgu95av2ENZYME2PROBE"
[17] "hgu95av2GENENAME"       "hgu95av2GO"
[19] "hgu95av2GO2ALLPROBES"   "hgu95av2GO2PROBE"
[21] "hgu95av2MAP"            "hgu95av2MAPCOUNTS"
[23] "hgu95av2OMIM"           "hgu95av2ORGANISM"
[25] "hgu95av2PATH"           "hgu95av2PATH2PROBE"
[27] "hgu95av2PFAM"           "hgu95av2PMID"
[29] "hgu95av2PMID2PROBE"     "hgu95av2PROSITE"
[31] "hgu95av2REFSEQ"         "hgu95av2SYMBOL"
[33] "hgu95av2UNIGENE"        "hgu95av2UNIPROT"
```

As before, it is possible to call the package name as a function to get some QC information about it.

```
R> qcdata = capture.output(hgu95av2())
```

```
R> head(qcdata, 20)
```

```
[1] "Quality control information for hgu95av2:"
[2] ""
[3] ""
[4] "This package has the following mappings:"
[5] ""
[6] "hgu95av2ACCNUM has 12625 mapped keys (of 12625 keys)"
[7] "hgu95av2ALIAS2PROBE has 37934 mapped keys (of 37934 keys)"
[8] "hgu95av2CHR has 11957 mapped keys (of 12625 keys)"
[9] "hgu95av2CHRLNGTHS has 25 mapped keys (of 25 keys)"
[10] "hgu95av2CHRLC has 11789 mapped keys (of 12625 keys)"
[11] "hgu95av2CHRLCEND has 11789 mapped keys (of 12625 keys)"
```

```

[12] "hgu95av2ENSEMBL has 11639 mapped keys (of 12625 keys)"
[13] "hgu95av2ENSEMBL2PROBE has 9021 mapped keys (of 9021 keys)"
[14] "hgu95av2ENTREZID has 11960 mapped keys (of 12625 keys)"
[15] "hgu95av2ENZYME has 1978 mapped keys (of 12625 keys)"
[16] "hgu95av2ENZYME2PROBE has 725 mapped keys (of 725 keys)"
[17] "hgu95av2GENENAME has 11960 mapped keys (of 12625 keys)"
[18] "hgu95av2G0 has 11363 mapped keys (of 12625 keys)"
[19] "hgu95av2G02ALLPROBES has 9581 mapped keys (of 9581 keys)"
[20] "hgu95av2G02PROBE has 6774 mapped keys (of 6774 keys)"

```

Alternatively, you can get similar information on how many items are in each of the provided maps by looking at the MAPCOUNTS:

```
R> hgu95av2MAPCOUNTS
```

To demonstrate the *environment* API, we'll start with a random sample of probe set IDs.

```

R> all_probes <- ls(hgu95av2ENTREZID)
R> length(all_probes)

[1] 12625

R> set.seed(0xa1beef)
R> probes <- sample(all_probes, 5)
R> probes

[1] "31882_at"   "38780_at"   "37033_s_at" "1702_at"    "31610_at"

```

The usual ways of accessing annotation data are also available.

```

R> hgu95av2ENTREZID[[probes[1]]]

[1] "9136"

R> hgu95av2ENTREZID$"31882_at"

[1] "9136"

R> syms <- unlist(mget(probes, hgu95av2SYMBOL))
R> syms

31882_at  38780_at 37033_s_at  1702_at  31610_at
"RRP9"   "AKR1A1" "GPX1"     "IL2RA" "PDZK1IP1"

```

2.0.5 Manipulating Bimap Objects

Many filtering operations on the annotation *environment* objects require conversion of the *environment* into a *list*. There is an `as.list` method for *AnnDbBimap* objects. In general, converting to lists will not be the most efficient way to filter the annotation data when using a SQLite-based package.

```
R> zz <- as.list(hgu95av2SYMBOL)
```

In an environment-based package, each mapping is its own object. To save disk and memory resources, not all reverse mappings are included in the environment-based packages. Here is the common idiom for creating a list that serves as the reverse map of a given environment.

```
R> library("hgu95av2", warn.conflicts=FALSE)
R> ## we load the environment so as not
R> ## to include the load time in the timing
R> class(hgu95av2SYMBOL)
R> system.time({
  p2sym <- as.list(hgu95av2SYMBOL)
  lens <- sapply(p2sym, length)
  nms <- rep(names(p2sym), lens)
  sym2p <- split(unlist(p2sym), nms)
})
R> ## in fact, there is a convenience function
R> ## for this operation in Biobase
R> system.time({
  p2sym <- as.list(hgu95av2SYMBOL)
  sym2p <- reverseSplit(p2sym)
})
R> detach("package:hgu95av2")
```

The SQLite-based package provide the same reverse maps as objects in the package name space for backwards compatibility, but the reverse mappings of any map is available using `revmap`. Since the data are stored as tables, no extra disk space is needed to provide reverse mappings.

```
R> system.time(sym2p <- revmap(hgu95av2SYMBOL))

  user  system elapsed
0.000   0.000   0.001
```

```
R> unlist(mget(syms, revmap(hgu95av2SYMBOL)))

      RRP9      AKR1A1      GPX1      IL2RA      PDZK1IP1
"31882_at"  "38780_at" "37033_s_at"  "1702_at"  "31610_at"
```

So now that you know about the `revmap` function you might try something like this:

```
R> as.list(revmap(hgu95av2PATH)["00300"])

$`00300`
[1] "34336_at" "35870_at" "35761_at"
```

But in the case of the `PATH` map, we don't need to use `revmap(x)` because `hgu95av2.db` already provides the `PATH2PROBE` map:

```
R> x <- hgu95av2PATH
R> ## except for the name, this is exactly revmap(x)
R> revx <- hgu95av2PATH2PROBE
R> revx2 <- revmap(x, objName="PATH2PROBE")
R> revx2
```

PATH2PROBE map for chip `hgu95av2` (object of class "AnnDbBimap")

```
R> identical(revx, revx2)

[1] TRUE

R> as.list(revx["00300"])

$`00300`
[1] "34336_at" "35870_at" "35761_at"
```

2.0.6 Displaying the Contents and Structure or Bimap Objects

Sometimes you may just want to know what elements are in an individual map. A *Bimap* interface is available to access the data in table (*data.frame*) format using `[` and `toTable`.

```
R> toTable(hgu95av2GO[probes[1:3]])
```

	probe_id	go_id	Evidence	Ontology
1	31882_at	GO:0006364	TAS	BP
2	37033_s_at	GO:0001659	IEA	BP
3	37033_s_at	GO:0001836	IMP	BP
4	37033_s_at	GO:0001885	IEA	BP
5	37033_s_at	GO:0002862	IEA	BP
6	37033_s_at	GO:0055114	IEA	BP
7	37033_s_at	GO:0006749	IDA	BP
8	37033_s_at	GO:0006629	IEA	BP
9	37033_s_at	GO:0006641	IEA	BP
10	37033_s_at	GO:0006916	IMP	BP
11	37033_s_at	GO:0006917	IEA	BP
12	37033_s_at	GO:0007605	IEA	BP
13	37033_s_at	GO:0008631	IEA	BP
14	37033_s_at	GO:0009636	IEA	BP
15	37033_s_at	GO:0009410	IEA	BP
16	37033_s_at	GO:0009609	IEA	BP
17	37033_s_at	GO:0009611	IEA	BP
18	37033_s_at	GO:0009650	IMP	BP
19	37033_s_at	GO:0010269	IMP	BP
20	37033_s_at	GO:0010332	IEA	BP
21	37033_s_at	GO:0014902	IEA	BP
22	37033_s_at	GO:0018158	IEA	BP
23	37033_s_at	GO:0033194	IEA	BP
24	37033_s_at	GO:0033599	IMP	BP
25	37033_s_at	GO:0040029	IDA	BP
26	37033_s_at	GO:0042311	IEA	BP
27	37033_s_at	GO:0042744	IEA	BP
28	37033_s_at	GO:0043154	IMP	BP
29	37033_s_at	GO:0043403	IEA	BP
30	37033_s_at	GO:0043523	IEA	BP
31	37033_s_at	GO:0043534	IEA	BP
32	37033_s_at	GO:0045444	IEA	BP
33	37033_s_at	GO:0051450	IEA	BP
34	37033_s_at	GO:0051702	IEA	BP
35	37033_s_at	GO:0051897	IEA	BP
36	37033_s_at	GO:0060047	IEA	BP
37	37033_s_at	GO:0060055	IEA	BP
38	38780_at	GO:0006006	TAS	BP
39	38780_at	GO:0055114	IEA	BP

40	38780_at	G0:0019853	IEA	BP
41	38780_at	G0:0042840	IEA	BP
42	38780_at	G0:0046185	IEA	BP
43	31882_at	G0:0030532	TAS	CC
44	31882_at	G0:0005634	IEA	CC
45	31882_at	G0:0005732	TAS	CC
46	37033_s_at	G0:0005739	IEA	CC
47	37033_s_at	G0:0005829	EXP	CC
48	37033_s_at	G0:0005737	IEA	CC
49	38780_at	G0:0005829	IEA	CC
50	38780_at	G0:0016324	IEA	CC
51	31882_at	G0:0003723	IEA	MF
52	37033_s_at	G0:0004602	EXP	MF
53	37033_s_at	G0:0004602	IEA	MF
54	37033_s_at	G0:0016491	IEA	MF
55	37033_s_at	G0:0008430	IEA	MF
56	37033_s_at	G0:0008539	IDA	MF
57	37033_s_at	G0:0017124	IPI	MF
58	37033_s_at	G0:0043295	IC	MF
59	38780_at	G0:0009055	TAS	MF
60	38780_at	G0:0005515	IPI	MF
61	38780_at	G0:0004032	TAS	MF
62	38780_at	G0:0016491	IEA	MF
63	38780_at	G0:0008106	IEA	MF
64	38780_at	G0:0047939	IEA	MF

The `toTable` function will display all of the information in a *Bimap*. This includes both the left and right values along with any other attributes that might be attached to those values. The left and right keys of the *Bimap* can be extracted using `Lkeys` and `Rkeys`. If it is necessary to only display information that is directly associated with the left to right links in a *Bimap*, then the `links` function can be used. The `links` returns a data frame with one row for each link in the bimap that it is applied to. It only reports the left and right keys along with any attributes that are attached to the edge between these two values.

Note that the order of the cols returned by `toTable` does not depend on the direction of the map ("undirected method"):

```
R> toTable(x)[1:6, ]
```

```

      probe_id path_id
1 38187_at    00232
2 38187_at    00983
3 38912_at    00232
4 38912_at    00983
5 36512_at    00960
6 36332_at    00380

```

```
R> toTable(revx)[1:6, ]
```

```

      probe_id path_id
1 38187_at    00232
2 38187_at    00983
3 38912_at    00232
4 38912_at    00983
5 36512_at    00960
6 36332_at    00380

```

Note however that the Lkeys are always on the left (1st col), the Rkeys always in the 2nd col

There can be more than 2 columns in the returned data frame:
3 cols:

```
R> toTable(hgu95av2PFAM)[1:6, ] # the right values are tagged
```

```

      probe_id      ipi_id PfamId
1 1000_at IPI00018195 PF00069
2 1000_at IPI00304111 PF00069
3 1000_at IPI00742900 PF00069
4 1000_at IPI00793141 PF00069
5 1001_at IPI00019530 PF00041
6 1001_at IPI00019530 PF00047

```

```
R> as.list(hgu95av2PFAM["1000_at"])
```

```

$`1000_at`
IPI00018195 IPI00304111 IPI00742900 IPI00793141
"PF00069"   "PF00069"   "PF00069"   "PF00069"

```

But the Rkeys are ALWAYS in the 2nd col.

For length() and keys(), the result does depend on the direction ("directed methods"):

```
R> length(x)
```

```
[1] 12625
```

```
R> length(revx)
```

```
[1] 203
```

```
R> allProbeSetIds <- keys(x)
```

```
R> allKEGGIds <- keys(revx)
```

There are more "undirected" methods listed below:

```
R> junk <- Lkeys(x)           # same for all maps in hgu95av2.db (except pseudo-map
```

```
R>                               # MAPCOUNTS)
```

```
R> Llength(x)                # nb of Lkeys
```

```
[1] 12625
```

```
R> junk <- Rkeys(x)          # KEGG ids for PATH/PATH2PROBE maps, GO ids for
```

```
R>                               # GO/GO2PROBE/GO2ALLPROBES maps, etc...
```

```
R> Rlength(x)                # nb of Rkeys
```

```
[1] 203
```

Notice how they give the same result for `x` and `revmap(x)`

2.0.7 Advantages of using revmap

Using `revmap` can be very efficient in some use cases:

```
R> x <- hgu95av2CHR
```

```
R> Rkeys(x)
```

```
[1] "1" "10" "11" "12" "13" "14" "15" "16" "17" "18" "19" "2" "20"
```

```
[14] "21" "22" "3" "4" "5" "6" "7" "8" "9" "Un" "X" "Y"
```

```
R> chroms <- Rkeys(x)[23:24]
```

```
R> chroms
```

```
[1] "Un" "X"
```

```
R> Rkeys(x) <- chroms
```

```
R> toTable(x)[1:10, ]
```

	probe_id	chromosome
1	41681_at	X
2	33496_at	X
3	39970_at	X
4	37285_at	X
5	33603_at	X
6	34588_i_at	X
7	34589_f_at	X
8	37740_r_at	X
9	40435_at	X
10	40436_g_at	X

To get this in the classic named-list format:

```
R> z <- as.list(revmap(x)[chroms])
R> names(z)
```

```
[1] "Un" "X"
```

```
R> z[["Y"]][1:5]
```

```
NULL
```

Compare to what you need to do this with the current envir-based package:

```
R> library(hgu95av2)
R> u <- unlist(as.list(hgu95av2CHR))
R> u <- u[u %in% chroms]
R> split(names(u), u)
```

A last example with cytogenetic locations:

```
R> x <- hgu95av2MAP
R> toTable(hgu95av2MAP)[1:6, ]
```

	probe_id	cytogenetic_location
1	38187_at	8p23.1-p21.3
2	38912_at	8p22
3	33825_at	14q32.1
4	36512_at	3q21.3-q25.2
5	38434_at	2q35
6	36332_at	17q25

```
R> as.list(revmap(x) ["8p22"])
```

```
$`8p22`
```

```
[1] "38912_at" "32372_at" "41416_at" "41209_at" "39981_at"
[6] "39982_r_at" "36850_at" "36851_g_at" "36852_at" "34553_at"
[11] "37363_at" "37951_at" "38013_at"
```

Are the probes in 'pbids' mapped to cytogenetic location "6p21.3"?

```
R> pbids <- c("38912_at", "41654_at", "907_at", "2053_at", "2054_g_at",
             "40781_at")
```

```
R> x <- subset(x, Lkeys=pbids, Rkeys="18q11.2")
```

```
R> toTable(x)
```

```
  probe_id cytogenetic_location
1  2053_at                18q11.2
2 2054_g_at                18q11.2
```

To coerce this map to a named vector:

```
R> pb2cyto <- as.character(x)
```

```
R> pb2cyto[pbids]
```

```
  <NA>      <NA>      <NA>  2053_at 2054_g_at      <NA>
      NA       NA       NA  "18q11.2" "18q11.2"      NA
```

The coercion of the reverse map works too but issues a warning because of the duplicated names:

```
R> cyto2pb <- as.character(revmap(x))
```

2.0.8 Using SQL to speed things up

Another area where the SQLite-based packages provide some advantages is when one wishes to filter the available annotation data in a complex fashion. For example, consider the task of obtaining all gene symbols on which are probed on a chip that have at least one GO BP ID annotation with evidence code IMP, IGI, IPI, or IDA. Here is one way to extract this using the environment-based packages:

```

R> ## Obtain SYMBOLS with at least one GO BP
R> ## annotation with evidence IMP, IGI, IPI, or IDA.
R> probes <- sample(all_probes, 500)
R> library("hgu95av2", warn.conflicts=FALSE)
R> system.time({
  bpids <- eapply(hgu95av2GO, function(x) {
    if (length(x) == 1 && is.na(x))
      NA
    else {
      sapply(x, function(z) {
        if (z$Ontology == "BP")
          z$GOID
        else
          NA
      })
    }
  })
  bpids <- unlist(bpids)
  bpids <- unique(bpids[!is.na(bpids)])
  g2p <- mget(bpids, hgu95av2GO2PROBE)
  wantedp <- lapply(g2p, function(x) {
    x[names(x) %in% c("IMP", "IGI", "IPI", "IDA")]
  })
  wantedp <- wantedp[sapply(wantedp, length) > 0]
  wantedp <- unique(unlist(wantedp))
  ans <- unlist(mget(wantedp, hgu95av2SYMBOL))
})
R> detach("package:hgu95av2")
R> length(ans)
R> ans[1:10]

```

All of the above code could have been reduced to a single SQL query with the SQLite-based packages. But to put together this query, you would need to look 1st at the schema to know what tables are present:

```
R> hgu95av2_dbschema()
```

This function will give you an output of all the create table statements that were used to generate the hgu5av2 database. Then you could assemble the sql query and use the helper function `hgu95av2_dbconn` to get a connection object for the database as follows:

```
R> system.time({
  SQL <- "SELECT symbol FROM go_bp INNER JOIN gene_info USING(_id)
        WHERE go_bp.evidence in ('IPI', 'IDA', 'IMP', 'IGI')"
  zz <- dbGetQuery(hgu95av2_dbconn(), SQL)
})

      user system elapsed
0.052  0.000  0.051
```

2.0.9 Combining data from separate annotation packages

Sometimes a user may wish to combine data that is found in two different annotation packages. One nice thing about the new packages is a side benefit of the fact that they are sqlite databases. This means that they can be attached into the same session, allowing easy joining of tables across otherwise separate databases. Being able to select items from multiple tables requires that their be a common value that can be used to identify those entries which are identical. It is also important to note that the internal IDs used in the *AnnotationDbi* packages cannot be used to map between packages since they have no meaning outside of the databases where they are defined.

In this example, we will join tables from *hgu95av2.db* and *GO.db*. To do this, we will attach the GO database to the HGU95-Av2 database to allow access to tables from both databases. We can then use GO identifiers as the link across the two data packages to create the join. In this section we are using the term *attach* to mean attaching using the SQL function ATTACH, and not the R function, or concept, of attaching. Before we begin, it is important to understand a little about where the GO database is located and its name. We use this information with the `system.file` function to construct a path to that database. In contrast, the *hgu95av2.db* package is already attached and we can use our predefined AnnotationDbi connection to it, `hgu95av2_dbconn()` to pass the SQL query that will attach the other database.

```
R> goDBLoc = system.file("extdata", "GO.sqlite", package="GO.db")
R> attachSQL = paste("ATTACH '", goDBLoc, "' as goDB;", sep = "")
R> dbGetQuery(hgu95av2_dbconn(), attachSQL)
```

NULL

Next, we are going to select some data, based on the GO ID, from two tables, one table from the HGU95-Av2 database and one from the GO

database. For brevity of output we will limit the query to 10 values. The WHERE clause on the last line of the SQL query specifies that the GO identifiers are the same. The first five lines of the query set up what variables to extract and what they will be named in the output.

```
R> selectSQL = paste("SELECT DISTINCT a.go_id AS 'hgu95av2.go_id',"
                    "a._id AS 'hgu95av2._id',"
                    "g.go_id AS 'GO.go_id', g._id AS 'GO._id',"
                    "g.ontology",
                    "FROM go_bp_all AS a, goDB.go_term AS g",
                    "WHERE a.go_id = g.go_id LIMIT 10;")
R> dataOut = dbGetQuery(hgu95av2_dbconn(), selectSQL)
R> dataOut
```

	hgu95av2.go_id	hgu95av2._id	GO.go_id	GO._id	ontology
1	GO:0000002	255	GO:0000002	16	BP
2	GO:0000002	1611	GO:0000002	16	BP
3	GO:0000002	3733	GO:0000002	16	BP
4	GO:0000002	4593	GO:0000002	16	BP
5	GO:0000003	41	GO:0000003	17	BP
6	GO:0000003	43	GO:0000003	17	BP
7	GO:0000003	81	GO:0000003	17	BP
8	GO:0000003	83	GO:0000003	17	BP
9	GO:0000003	105	GO:0000003	17	BP
10	GO:0000003	106	GO:0000003	17	BP

```
R> #just to clean up we can now detach the GO database...
R> detachSQL = paste("DETACH goDB")
R> dbGetQuery(hgu95av2_dbconn(), detachSQL)
```

NULL

This query combines the `go_bp_all` table from the HGU95-Av2 database with the `go_term` table from the GO database. They are joined based on their `go_id` columns. For illustration purposes, the internal ID `_id` and the `go_id` from both tables are included in the output. This demonstrates that the `go_ids` can be used to join these tables while the internal IDs cannot. The internal IDs, `_id`, are suitable for joins within a single database, but cannot be used across databases.