Using the new Annotation Packages

Marc Carlson

29 July 2008

1 Introduction

The older style of annotation packages used simple environments to represent data in a searchable way. But there were some drawbacks. More data could not be added to a package without creating more environments which sometimes resulted in duplicated information. But even more importantly, the data was not packaged in a format that allowed for efficient re-purposing. Because it is impossible to predict what questions will arise, this lack of flexibility is a major problem for data that is needed for scientific inquiry. While the older packages might be able to address questions for a handful of information types today, as more and more kinds of information get added, it becomes increasingly difficult to guess in advance all of the ways that people will need to use it. To address this the newer style of annotation packages contain real databases within them. The databases used are lightweight SQLite databases, which are modular and can be connected to each other on the fly to allow data to be re-purposed not only within a database, but also between databases.

In order to provide backwards compatibility and convenience, the new system allows access from two different levels. Users can use mappings which are generated on the fly to emulate the old style environments, or they can access the databases with direct SQL queries. The emulated mappings are provided by the AnnotationDbi package, which is built on packages that already allow direct access to the database. Meanwhile the schemas for the databases created by using the SQLForge functions in AnnotationDbi have been designed with a consistent pattern in order to allow easy reuse of the data. This lab will go through a series of examples to illustrate how these new annotation packages work and how users can take advantage of both kinds of access to get the most out of the new annotation system.

2 The contents of an annotation package

Lets begin by loading a typical annotation package

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

Notice all that stuff that loaded along with it? A lot of that stuff is here to to allow us to interface with the database. DBI and RSQLite allow R to talk to SQLite databases, and AnnotationDbi helps by emulating the old style annotation environments

Now lets look at what sorts of things get made whenever we load the package:

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

```
 [1] "hgu95av2"               "hgu95av2ACCNUM"         "hgu95av2ALIAS2PROBE"
 [4] "hgu95av2CHR"            "hgu95av2CHRLENGTHS"     "hgu95av2CHRLOC"
 [7] "hgu95av2_dbconn"        "hgu95av2_dbfile"        "hgu95av2_dbInfo"
[10] "hgu95av2_dbschema"      "hgu95av2ENSEMBL"        "hgu95av2ENSEMBL2PROBE"
[13] "hgu95av2ENTREZID"       "hgu95av2ENZYME"         "hgu95av2ENZYME2PROBE"
[16] "hgu95av2GENENAME"       "hgu95av2GO"             "hgu95av2GO2ALLPROBES"
[19] "hgu95av2GO2PROBE"       "hgu95av2MAP"            "hgu95av2MAPCOUNTS"
[22] "hgu95av2OMIM"           "hgu95av2ORGANISM"       "hgu95av2PATH"
[25] "hgu95av2PATH2PROBE"     "hgu95av2PFAM"           "hgu95av2PMID"
[28] "hgu95av2PMID2PROBE"     "hgu95av2PROSITE"        "hgu95av2REFSEQ"
[31] "hgu95av2SYMBOL"         "hgu95av2UNIGENE"
```

As before, each package has a function named after it that can be called to get some QC information about it.

```
> qcdata <- capture.output(hgu95av2())
> 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 36833 mapped keys (of 36833 keys)"
 [8] "hgu95av2CHR has 12117 mapped keys (of 12625 keys)"
 [9] "hgu95av2CHRLENGTHS has 25 mapped keys (of 25 keys)"
[10] "hgu95av2CHRLOC has 11817 mapped keys (of 12625 keys)"
[11] "hgu95av2ENSEMBL has 11156 mapped keys (of 12625 keys)"
[12] "hgu95av2ENSEMBL2PROBE has 8286 mapped keys (of 8286 keys)"
[13] "hgu95av2ENTREZID has 12124 mapped keys (of 12625 keys)"
[14] "hgu95av2ENZYME has 1957 mapped keys (of 12625 keys)"
[15] "hgu95av2ENZYME2PROBE has 709 mapped keys (of 709 keys)"
[16] "hgu95av2GENENAME has 12124 mapped keys (of 12625 keys)"
[17] "hgu95av2GO has 11602 mapped keys (of 12625 keys)"
[18] "hgu95av2GO2ALLPROBES has 8383 mapped keys (of 8383 keys)"
[19] "hgu95av2GO2PROBE has 5898 mapped keys (of 5898 keys)"
[20] "hgu95av2MAP has 12093 mapped keys (of 12625 keys)"
```

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

```
> hgu95av2MAPCOUNTS
```

```
       hgu95av2ACCNUM   hgu95av2ALIAS2PROBE           hgu95av2CHR
                12625                 36833                 12117
   hgu95av2CHRLENGTHS       hgu95av2CHRLOC       hgu95av2ENSEMBL
                   25                 11817                 11156
hgu95av2ENSEMBL2PROBE     hgu95av2ENTREZID       hgu95av2ENZYME
                 8286                 12124                  1957
 hgu95av2ENZYME2PROBE     hgu95av2GENENAME            hgu95av2GO
                  709                 12124                 11602
 hgu95av2GO2ALLPROBES     hgu95av2GO2PROBE           hgu95av2MAP
                 8383                  5898                 12093
         hgu95av2OMIM         hgu95av2PATH    hgu95av2PATH2PROBE
                10390                  4415                   199
         hgu95av2PFAM         hgu95av2PMID    hgu95av2PMID2PROBE
                12048                 12097                178969
      hgu95av2PROSITE        hgu95av2REFSEQ        hgu95av2SYMBOL
                12048                 12011                 12124
      hgu95av2UNIGENE
                12097
```

It can also sometimes be useful to quickly look at some of the metadata from the package. The following will give information about the database and the sources that were used to make the hgu95av2 package:

```
> hgu95av2_dbInfo()[c(1:4, 8:10), ]
```

```
              name                                value
1  DBSCHEMAVERSION                                  1.0
2         DBSCHEMA                           HUMANCHIP_DB
3         ORGANISM                           Homo sapiens
4          SPECIES                                  Human
8     EGSOURCEDATE                              2008-Apr2
9     EGSOURCENAME                           Entrez Gene
10     EGSOURCEURL ftp://ftp.ncbi.nlm.nih.gov/gene/DATA
```

Because these packages are based on SQLite databases underneath, it can also sometimes be useful to know what is in there. The 1st thing we give you for this is a handle to the database:

```
> con <- hgu95av2_dbconn()
```

This handle lets you do things like list all the database tables:

```
> dbListTables(con)
```

```
 [1] "alias"                 "chrlengths"            "chromosome_locations"
 [4] "chromosomes"           "cytogenetic_locations" "ec"
 [7] "ensembl"               "gene_info"             "genes"
[10] "go_bp"                 "go_bp_all"             "go_cc"
[13] "go_cc_all"             "go_mf"                 "go_mf_all"
[16] "kegg"                  "map_counts"            "map_metadata"
[19] "metadata"              "omim"                  "pfam"
[22] "probes"                "prosite"               "pubmed"
[25] "refseq"                "sqlite_stat1"          "unigene"
```

Or list all the fields from one such table:

```
> dbListFields(con, "map_metadata")

[1] "map_name"    "source_name" "source_url"  "source_date"
```

You can also use this handle to get data straight out of the database by submitting SQL queries. Here is a simple teaser example of how you can do this:

```
> sql <- "SELECT * FROM map_metadata;"
> dbGetQuery(con, sql)[c(1, 12, 20, 23), ]

      map_name                                source_name
1     ENTREZID                              Entrez Gene
12          GO                            Gene Ontology
20      CHRLOC UCSC Genome Bioinformatics (Homo sapiens)
23 ALIAS2PROBE                              Entrez Gene
                                                      source_url
1                                  ftp://ftp.ncbi.nlm.nih.gov/gene/DATA
12          ftp://ftp.geneontology.org/pub/go/godatabase/archive/latest
20 ftp://hgdownload.cse.ucsc.edu/goldenPath/currentGenomes/Homo_sapiens
23                                 ftp://ftp.ncbi.nlm.nih.gov/gene/DATA
   source_date
1    2008-Apr2
12      200803
20   2006-Apr14
23   2008-Apr2
```

You will notice that this table has data that is very similar to the output of hgu95av2_dbInfo(), so now you know where hgu95av2_dbInfo() gets its information from.

3 A brief SQL primer

SQL is worth learning. Not only is it very powerful, it is also dead simple to start using. SQL is very declarative and so a simple example is easy to "read".

Consider the following SQL statement:

SELECT * FROM table;

This statement will retrieve all (indicated by the wildcard *) of the entries from ”table”. Not too hard. But we can embellish that previous statement with a WHERE clause like this:

SELECT * FROM table WHERE field=”bar”;

This statement just adds a filter to the content returned to by the statement. Thats still easy to read. But WHERE clauses can make things really complicated in a hurry. Not only can WHERE clauses be used to filter, but they can also be used to join information in different tables by requiring that two keys match. This can result in queries that start to look more like this:

SELECT field1,field2
FROM table1,table2
WHERE table1.field1=table2.field2;

Which is still pretty simple, but I think you can see why this can get messy. This is why the AS keyword was created. The AS keyword just lets you give an alias to a part of your query so that you can write things more clearly. Here how the previous example would read once you put an alias on both the table-names.

SELECT field1,field2
FROM table1 AS T1, table2 AS T2
WHERE T1.field1=T2.field2;

Now that we have had a look at a little bit of SQL, lets try another select from the database:

```
> sql <- "SELECT * FROM alias LIMIT 10;"
> dbGetQuery(con, sql)

   _id alias_symbol
1    4         AAC1
2    4         NATI
3    4         NAT1
4    5         AAC2
5    5         NAT2
6    7         AACT
7    7          ACT
8    7        GIG24
9    7        GIG25
10   7     MGC88254
```

And another:

```
> sql <- "SELECT * FROM genes LIMIT 10;"
> dbGetQuery(con, sql)
```

```
   _id gene_id
1    4       9
2    5      10
3    7      12
4    8      13
5    9      14
6   10      15
7   11      16
8   13      18
9   14      19
10  15      20
```

You will notice that one of the fields in these last couple of tables is _id. This is not a coincidence, and it is actually quite important because _id is an important internal key for almost all of the tables in these databases. It is important to be aware however that _id has no meaning outside of a given database. You cannot join on _id outside of a database. To join across databases, you need to use a different key.

EXERCISE 1: Query one of the other database tables.

4 Using the new annotation mappings

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

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

```
[1] 12625
```

```
> set.seed(10600175)
> probes <- sample(all_probes, 5)
> probes
```

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

Now that we have some probes to test with, we can demonstrate that the usual ways of accessing annotation data are also available.

```
> hgu95av2ENTREZID[[probes[1]]]
```

```
[1] "9136"

> hgu95av2ENTREZID$"31882_at"

[1] "9136"

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

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

In terms of the underlying database, most mappings are simple joins between two tables. You may recall the _id field that is present in most of the tables. This _id is what is used for the internal joins to make each mapping. To demonstrate, here is an example of the simple join used to make the symbol mapping described above:

```
> sql <- paste("SELECT probes.probe_id, gene_info.symbol",
+              "FROM probes, gene_info WHERE probes._id = gene_info._id",
+              "AND gene_info.symbol NOT NULL;")
> table <- dbGetQuery(con, sql)
> head(table)

  probe_id   symbol
1 38187_at     NAT1
2 38912_at     NAT2
3 33825_at SERPINA3
4 36512_at    AADAC
5 38434_at     AAMP
6 36332_at    AANAT
```

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.

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

In the older style environment-based packages, each mapping was its own object. To save disk and memory resources, not all reverse mappings were included in these older environment-based packages.

For compatibility, the newer SQLite-based packages provide the same reverse maps as were available before. But now the reverse mappings of any map can be made instantly available as well. This done with the `revmap()` function. Since the data are stored as tables in the internal database, no extra disk space is needed to provide reverse mappings.

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

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

> 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:

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

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

But in the case of the PATH map, we don't actually need to use `revmap()` because hgu95av2.db already provides the PATH2PROBE map. In fact, we can demonstrate that what is made by `revmap()` is the same as the default reverse mapping.

```
> revPATH <- hgu95av2PATH2PROBE
> revPATH

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

> revPATH2 <- revmap(hgu95av2PATH, objName = "PATH2PROBE")
> revPATH2

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

> identical(revPATH, revPATH2)

[1] TRUE

> as.list(revPATH["00300"])

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

EXERCISE 2: Use `revmap()` to find out which probes map to the gene symbol "HOXA9".

5 Displaying the Contents and Structure of Bimap Objects

Sometimes you may just need to know a little bit more about what elements are in an individual mapping. An example might be if you were wondering whether or not a map is reversible. Most maps are simple enough to be reversible, but some are more complex and reversing them would result in information loss. A *Bimap* interface is available to access the data as a table (*data.frame*) format using `[` and `toTable()`.

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

```
    probe_id      go_id Evidence Ontology
1   31882_at GO:0006364      TAS       BP
2 37033_s_at GO:0001836      IMP       BP
3 37033_s_at GO:0006749      IDA       BP
4 37033_s_at GO:0006916      IMP       BP
5 37033_s_at GO:0008631      IEA       BP
6 37033_s_at GO:0009650      IMP       BP
```

Each *Bimap* has a set of left and right keys that it maps. Between each pair of mapped keys, there is an edge. Additional information can be attached to the edge of a bimap or to either of the keys as attributes. In the case of the mapping represented above, the evidence code is an attribute that is attached to each edge, because this information is affiliated with the relationship between the Lkey and the Rkey. In contrast, the Ontology information is an attribute that is attached to the Rkey only.

As you can see, the `toTable()` function will display all of the information in a *Bimap*. This includes both the left and right key values along with any other attributes that might be attached to those values or to the edges between those values. The left and right keys of the *Bimap* can aslo be extracted using `Lkeys()` and `Rkeys()`.

```
> Lkeys(hgu95av2GO[probes[1:3]])
```

```
[1] "31882_at"   "38780_at"   "37033_s_at"
```

```
> head(Rkeys(hgu95av2GO[probes[1:3]]))
```

```
[1] "GO:0008152" "GO:0006953" "GO:0006954" "GO:0019216" "GO:0006928"
[6] "GO:0007623"
```

If is 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 Lkey to Rkey link in the bimap that it is applied to. The `links()` function only reports the left and right keys along with any attributes that are attached to the edge between these two values.

```
> head(links(hgu95av2GO[probes[1:3]]))

    probe_id      go_id Evidence
1   31882_at GO:0006364      TAS
2 37033_s_at GO:0001836      IMP
3 37033_s_at GO:0006749      IDA
4 37033_s_at GO:0006916      IMP
5 37033_s_at GO:0008631      IEA
6 37033_s_at GO:0009650      IMP
```

Notice that the results returned by `toTable()` does not depend on the direction of the map. The output the Lkeys are always on the left (1st col), and the Rkeys are always in the 2nd col. Thus, `toTable()` is an "undirected" method.

```
> toTable(hgu95av2PATH)[1:6, ]

  probe_id path_id
1 38187_at   00232
2 38912_at   00232
3 36512_at   00650
4 36512_at   00960
5 36332_at   00380
6 36185_at   00252

> toTable(revmap(hgu95av2PATH))[1:6, ]

  probe_id path_id
1 38187_at   00232
2 38912_at   00232
3 36512_at   00650
4 36512_at   00960
5 36332_at   00380
6 36185_at   00252
```

Here is another example of a map where `toTable()` returns a value that contains additional attributes, and thus the data frame contains 3 cols.

```
> toTable(hgu95av2PFAM)[1:6, ]

  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 PF07714
6  1001_at IPI00019530 PF00041

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

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

Notice that as before the Lkeys are the 1st col and the Rkeys are ALWAYS in the 2nd col.

In contrast to `toTable()`, for the functions `length()` and `keys()`, the result returned does depend on the direction of the input map. Thus, these are directed methods:

```
> length(hgu95av2PATH)

[1] 12625

> length(revmap(hgu95av2PATH))

[1] 199

> allProbeSetIds <- keys(hgu95av2PATH)
> allProbeSetIds[1:3]

[1] "1000_at"   "1001_at"   "1002_f_at"

> allKEGGIds <- keys(revmap(hgu95av2PATH))
> allKEGGIds[1:3]

[1] "00232" "00650" "00960"
```

The functions `Llength()` and `Rlength()` are both handy for if you just want to know how many Lkeys or Rkeys you have.

```
> Llength(hgu95av2PATH)

[1] 12625

> length(Lkeys(hgu95av2PATH))

[1] 12625

> Rlength(hgu95av2PATH)

[1] 199

> length(Rkeys(hgu95av2PATH))

[1] 199
```

Notice how they give the same result for hgu95av2PATH and revmap(hgu95av2PATH)? The `Llength()` and `Rlength()` are therefore both undirected methods.

As a specific example, here is how we can ask whether the probes in 'pbids' are mapped to cytogenetic location "6p21.3"?

```
> pbids <- c("38912_at", "41654_at", "907_at", "2053_at", "2054_g_at",
+     "40781_at")
> regionalSubset <- subset(hgu95av2MAP, Lkeys = pbids, Rkeys = "18q11.2")
> toTable(regionalSubset)

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

To coerce this map into a named vector:

```
> pb2cyto <- as.character(regionalSubset)
> 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:

```
> cyto2pb <- as.character(revmap(regionalSubset))
```

EXERCISE 3: How could you used `mget()` to learn all possible aliased gene symbols for pbids? And how could you use `subset()` to find out which of these probesets are="ADA"?

6 The Organism and Inparanoid packages

Sometimes you will have a list of genes from one species and wonder what the homologous equivalent would be in another species. At other times you will need some information that is maybe a little bit more exotic than what you find in a standard chip package. Bioconductor provides mappings in the Inparanoid packages to try and assist with the homology problem while putting more exotic mappings into the organism level packages. Below is an example of how you can use the mappings in the inparanoid packages along with supplementary mappings in the organism packages to connect to a genes homolog in another species.

```
> library("hom.Hs.inp.db")
> ls("package:hom.Hs.inp.db")

 [1] "hom.Hs.inp"          "hom.Hs.inpAEDAE"     "hom.Hs.inpANOGA"
 [4] "hom.Hs.inpAPIME"     "hom.Hs.inpARATH"     "hom.Hs.inpBOSTA"
 [7] "hom.Hs.inpCAEBR"     "hom.Hs.inpCAEEL"     "hom.Hs.inpCAERE"
[10] "hom.Hs.inpCANFA"     "hom.Hs.inpCANGL"     "hom.Hs.inpCIOIN"
[13] "hom.Hs.inpCRYNE"     "hom.Hs.inpDANRE"     "hom.Hs.inp_dbconn"
```

```
[16] "hom.Hs.inp_dbfile"   "hom.Hs.inp_dbInfo"   "hom.Hs.inp_dbschema"
[19] "hom.Hs.inpDEBHA"     "hom.Hs.inpDICDI"     "hom.Hs.inpDROME"
[22] "hom.Hs.inpDROPS"     "hom.Hs.inpENTHI"     "hom.Hs.inpESCCO"
[25] "hom.Hs.inpFUGRU"     "hom.Hs.inpGALGA"     "hom.Hs.inpGASAC"
[28] "hom.Hs.inpKLULA"     "hom.Hs.inpMACMU"     "hom.Hs.inpMAPCOUNTS"
[31] "hom.Hs.inpMONDO"     "hom.Hs.inpMUSMU"     "hom.Hs.inpORGANISM"
[34] "hom.Hs.inpORYSA"     "hom.Hs.inpPANTR"     "hom.Hs.inpRATNO"
[37] "hom.Hs.inpSACCE"     "hom.Hs.inpSCHPO"     "hom.Hs.inpTETNI"
[40] "hom.Hs.inpXENTR"     "hom.Hs.inpYARLI"

> library("org.Hs.eg.db")
> mget("MSX2", org.Hs.egSYMBOL2EG)

$MSX2
[1] "4488"

> mget("4488", org.Hs.egENSEMBLPROT)

$`4488`
[1] "ENSP00000239243"

> mget("ENSP00000239243", hom.Hs.inpMUSMU)

$ENSP00000239243
[1] "MGI:97169"

> library("org.Mm.eg.db")
> mget("MGI:97169", org.Mm.egMGI2EG)

$`MGI:97169`
[1] "17702"

> mget("17702", org.Mm.egSYMBOL)

$`17702`
[1] "Msx2"
```

EXERCISE 4: Now that you have seen how to map a gene from human to mouse, try mapping the gene "Shh" back from mouse to man.

7 Using SQL to get access to data that is not exposed via a mapping

The most commonly used things will normally be placed into mappings for easy access. But sometimes there will be more data that will be in the database which will not be in a standard mapping. But this data can still be obtained

using some simple SQL queries. One example of this is within the inparanoid packages. For the inparanoid packages, only some of the data is present within the mappings, but all of the inparanoid data is present in the database tables. So how could you access this other data?

```
> dbListTables(hom.Hs.inp_dbconn())

 [1] "aedes_aegypti"              "anopheles_gambiae"
 [3] "apis_mellifera"             "arabidopsis_thaliana"
 [5] "bos_taurus"                 "caenorhabditis_briggsae"
 [7] "caenorhabditis_elegans"     "caenorhabditis_remanei"
 [9] "candida_glabrata"           "canis_familiaris"
[11] "ciona_intestinalis"         "cryptococcus_neoformans"
[13] "danio_rerio"                "debaryomyces_hanseneii"
[15] "dictyostelium_discoideum"   "drosophila_melanogaster"
[17] "drosophila_pseudoobscura"   "entamoeba_histolytica"
[19] "escherichia_coliK12"        "gallus_gallus"
[21] "gasterosteus_aculeatus"     "kluyveromyces_lactis"
[23] "macaca_mulatta"             "map_counts"
[25] "map_metadata"               "metadata"
[27] "monodelphis_domestica"      "mus_musculus"
[29] "oryza_sativa"               "pan_troglodytes"
[31] "rattus_norvegicus"          "saccharomyces_cerevisiae"
[33] "schizosaccharomyces_pombe" "sqlite_stat1"
[35] "takifugu_rubripes"          "tetraodon_nigroviridis"
[37] "xenopus_tropicalis"         "yarrowia_lipolytica"

> dbListFields(hom.Hs.inp_dbconn(), "entamoeba_histolytica")

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

> sql <- "SELECT * FROM entamoeba_histolytica LIMIT 10;"
> dbGetQuery(hom.Hs.inp_dbconn(), sql)

            inp_id clust_id species  score seed_status
1  ENSP00000371526        1   HOMSA      1        100%
2        XP_653245        1   ENTHI      1        100%
3  ENSP00000358400        2   HOMSA      1        100%
4        XP_654752        2   ENTHI      1        100%
5  ENSP00000228347        3   HOMSA      1        100%
6        XP_654890        3   ENTHI      1        100%
7  ENSP00000317123        4   HOMSA 0.0694
8  ENSP00000320252        4   HOMSA      1        100%
9        XP_655520        4   ENTHI      1        100%
10 ENSP00000263200        5   HOMSA 0.7613

> toTable(hom.Hs.inpENTHI)[1:10, ]
```

```
            inp_id  inp_id.1
1  ENSP00000371526 XP_653245
2  ENSP00000358400 XP_654752
3  ENSP00000228347 XP_654890
4  ENSP00000320252 XP_655520
5  ENSP00000269122 XP_650764
6  ENSP00000361446 XP_653748
7  ENSP00000354587 XP_655838
8  ENSP00000364811 XP_655936
9  ENSP00000264331 XP_651972
10 ENSP00000307940 XP_650528
```

EXERCISE 5: Query one of the tables and filter out entries that are not human and where the score = 1.

EXERCISE 6: Now use the fact that the clust_id is shared within this table between matching keys to try to recreate the data that is in the original inparanoid mapping.

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:

```
> probes <- sample(all_probes, 500)
> library("hgu95av2", warn.conflicts = FALSE)
> system.time({
+     bpids <- eapply(hgu95av2GO, function(hgu95av2PATH) {
+         if (length(hgu95av2PATH) == 1 && is.na(hgu95av2PATH))
+             NA
+         else {
+             sapply(hgu95av2PATH, 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(hgu95av2PATH) {
+         hgu95av2PATH[names(hgu95av2PATH) %in% c("IMP", "IGI",
+             "IPI", "IDA")]
+     })
+     wantedp <- wantedp[sapply(wantedp, length) > 0]
+     wantedp <- unique(unlist(wantedp))
+     ans <- unique(unlist(mget(wantedp, hgu95av2SYMBOL)))
+ })
> detach("package:hgu95av2")
> length(ans)
> sort(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 might need to look at what tables are present and what the fields are for the tables you want:

```
> dbListTables(hgu95av2_dbconn())

 [1] "alias"                "chrlengths"           "chromosome_locations"
 [4] "chromosomes"          "cytogenetic_locations" "ec"
 [7] "ensembl"              "gene_info"            "genes"
[10] "go_bp"                "go_bp_all"            "go_cc"
[13] "go_cc_all"            "go_mf"                "go_mf_all"
[16] "kegg"                 "map_counts"           "map_metadata"
[19] "metadata"             "omim"                 "pfam"
[22] "probes"               "prosite"              "pubmed"
[25] "refseq"               "sqlite_stat1"         "unigene"

> dbListFields(hgu95av2_dbconn(), "go_bp")

[1] "_id"      "go_id"    "evidence"

> dbListFields(hgu95av2_dbconn(), "gene_info")

[1] "_id"       "gene_name" "symbol"
```

If you need even more information then you can also look at:

```
> hgu95av2_dbschema()
```

This function will give you an output of all the create table statements that were used to generate the hgu5av2 database. Whichever method you use, you could then assemble a sql query to get the information directly from the database as follows:

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

```
+ SQLans <- dbGetQuery(hgu95av2_dbconn(), SQL)
+ SQLans <- unique(as.vector(t(SQLans)))
+ length(SQLans)
+ SQLans[1:10]
+ })

   user  system elapsed
  0.092   0.000   0.094
```

EXERCISE 7: How would you change the above SQL query to only look for evidence types of 'IPI' and 'IDA'?

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 remember that the internal _id's 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, we have to know 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.

```
> goDBLoc <- system.file("extdata", "GO.sqlite", package="GO.db")
> attachSQL <- paste("ATTACH '", goDBLoc, "' as goDB;", sep = "")
> 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. To keep things brief we will limit the query to 10 values. The `WHERE` clause on the last

line of the SQL query sqecifies 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.

```
> 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;")
> dataOut <- dbGetQuery(hgu95av2_dbconn(), selectSQL)
> dataOut

   hgu95av2.go_id hgu95av2._id   GO.go_id GO._id ontology
1      GO:0000002          255 GO:0000002     13       BP
2      GO:0000002         1633 GO:0000002     13       BP
3      GO:0000002         3804 GO:0000002     13       BP
4      GO:0000002         4680 GO:0000002     13       BP
5      GO:0000003           41 GO:0000003     14       BP
6      GO:0000003           43 GO:0000003     14       BP
7      GO:0000003           81 GO:0000003     14       BP
8      GO:0000003           83 GO:0000003     14       BP
9      GO:0000003          104 GO:0000003     14       BP
10     GO:0000003          105 GO:0000003     14       BP

> #just to clean up we can now detach the GO database...
> detachSQL <- paste("DETACH goDB")
> 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` collumns. 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.

EXERCISE 8: Can you join across a couple of databases? Make a join across the mapping in the hom.Hs.inp database to the ensemble_prot mapping in the org.Hs.eg.db database. This is an "advanced" query, so don't panic if you can't immediately figure this one out. The point is to get you a little more practiced.

10 Using SQLForge to make a custom package

In order to use SQLForge you really only need to have one kind of information and that is a list of paired IDs. These IDs are to be stored in a tab delimited file that is formatted in the same way that they used to be for the older AnnBuilder package. For those who are unfamiliar with the AnnBuilder package, this just means that there are two columns separated by a tab where the column on the left contains probe or probeset identifiers and the column on the right contains some sort of widely accepted gene accession. This file should NOT contain a header. SQLForge will then use these IDs along with it's own support databases to make an *AnnotationDbi* package for you. Here is how these IDs should look if you were to read them into R:

```
> read.table(system.file("extdata", "hcg110_ID", package = "AnnotationDbi"),
+     sep = "\t", header = FALSE, as.is = TRUE)[1:5, ]
```

```
         V1     V2
1   1000_at X60188
2   1001_at X60957
3 1002_f_at X65962
4 1003_s_at X68149
5   1004_at X68149
```

This example mapping is what SQLForge will use to define the package, but you will also need the latest data, to get that, you need to have the latest .db0 package installed:

```
> source("http://bioconductor.org/biocLite.R")
> biocLite("human.db0")
```

Wrapper functions will allow you to then make a database and package it up into a custom annotation package all at once. The following shows how:

```
> ## Because this is an example, we are going to use a tempdir()
> tmpout <- tempdir()
> ## Also need to get some IDs ready to pass in
> hcg110_IDs <- system.file("extdata",
+                           "hcg110_ID",
+                           package="AnnotationDbi")
> ## Then we just call the function
> makeHUMANCHIP_DB(affy=FALSE,
+     prefix="hcg110",
+     fileName=hcg110_IDs,
+     baseMapType="gb",
+     outputDir = tmpout,
+     version="1.0.0",
+     manufacturer = "Affymetrix",
+     chipName = "Human Cancer G110 Array",
+     manufacturerUrl = "http://www.affymetrix.com")
```

Wrapper functions are provided for making all of the different kinds of chip based package types that are presently defined. These are named after the schemas that they correspond to. So for example `makeHUMANCHIP_DB()` corresponds to the HUMANCHIP_DB schema, and is used to produce chip based annotation packages of that type.