AnnotationDbi

Marc Carlson, Herve Pages, Seth Falcon, Nianhua Li

March 20, 2010

1 Introduction

1.0.1 Purpose

AnnotationDbi is used primarily to create mapping objects that allow easy access from R to underlying annotation databases. As such, it acts as the R interface for all the standard annotation packages. Underlying each AnnotationDbi supported annotation package is at least one (and often two) annotation databases. AnnotationDbi also provides schemas for theses databases. For each supported model organism, a standard gene centric database is maintained from public sources and is packaged up as an appropriate organism or ”org” package.

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 package is currently using by using its ”_dbschema” method.

There is one schema/database in 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 schema Design of org packages

The current design of the organism packages is deliberately simple and 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. Each organism package is centered around one type of gene identifier. This identifier is found as the gene_id field in the genes table and is both the central ID for the database as well as the foreign key that chip packages should join to.

The chip packages are 'lightweight', and only contain information about the basic probe to gene mapping. You might wonder how such packages can provide access to all the other information that they do. This is possible because all the other data provided by chip packages comes from joins that are performed by AnnotationDbi behind the scenes at run time. All chip packages have a dependency on at least one organism package. The name of the organism package being depended on can be found by looking at its "ORGPKG" value. To learn about the schema from the appropriate organism package, you will need to look at the "_dbschema" method for that package. In the case of the chip packages, the gene_id that in these packages is mapped to the probe_ids, is used as a foreign key to the appropriate organism package.

Specialized packages like the packages for GO and KEGG, will have their own schemas but will also adhere to the use of an internal _id for joins between their tables. As with the organism packages, this _id is not suitable for use as a foreign key.

For a complete listing of the different schemas used by various packages, users can use the `available.dbschemas` function. This list will also tell you which model organisms are supported.

```
> require(org.Hs.eg.db)
> available.dbschemas()
```

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 they did with the older environment based annotation 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] "hgu95av2CHRLENGTHS"    "hgu95av2CHRLOC"
[11] "hgu95av2CHRLOCEND"     "hgu95av2ENSEMBL"
[13] "hgu95av2ENSEMBL2PROBE" "hgu95av2ENTREZID"
[15] "hgu95av2ENZYME"        "hgu95av2ENZYME2PROBE"
[17] "hgu95av2GENENAME"      "hgu95av2GO"
[19] "hgu95av2GO2ALLPROBES"  "hgu95av2GO2PROBE"
[21] "hgu95av2MAP"           "hgu95av2MAPCOUNTS"
[23] "hgu95av2OMIM"          "hgu95av2ORGANISM"
[25] "hgu95av2ORGPKG"        "hgu95av2PATH"
[27] "hgu95av2PATH2PROBE"    "hgu95av2PFAM"
[29] "hgu95av2PMID"          "hgu95av2PMID2PROBE"
[31] "hgu95av2PROSITE"       "hgu95av2REFSEQ"
[33] "hgu95av2SYMBOL"        "hgu95av2UNIGENE"
[35] "hgu95av2UNIPROT"
```

Exercise 1
Start an R session and use the `library` *function to load the hgu95av2.db software package. Use search() to see that an organism package was also loaded and then use the approriate "_dbschema" methods to the schema for the hgu95av2.db and org.Hs.eg.db packages.*

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 37697 mapped keys (of 109070 keys)"
 [8] "hgu95av2CHR has 11733 mapped keys (of 12625 keys)"
 [9] "hgu95av2CHRLENGTHS has 25 mapped keys (of 25 keys)"
[10] "hgu95av2CHRLOC has 11643 mapped keys (of 12625 keys)"
[11] "hgu95av2CHRLOCEND has 11643 mapped keys (of 12625 keys)"
[12] "hgu95av2ENSEMBL has 11552 mapped keys (of 12625 keys)"
[13] "hgu95av2ENSEMBL2PROBE has 8719 mapped keys (of 19892 keys)"
[14] "hgu95av2ENTREZID has 11736 mapped keys (of 12625 keys)"
[15] "hgu95av2ENZYME has 2028 mapped keys (of 12625 keys)"
[16] "hgu95av2ENZYME2PROBE has 737 mapped keys (of 901 keys)"
[17] "hgu95av2GENENAME has 11736 mapped keys (of 12625 keys)"
[18] "hgu95av2GO has 11309 mapped keys (of 12625 keys)"
[19] "hgu95av2GO2ALLPROBES has 10323 mapped keys (of 11236 keys)"
[20] "hgu95av2GO2PROBE has 7280 mapped keys (of 8245 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" 
```

The annotation packages provide a huge variety of information in each package. Some common types of information include gene symbols (SYMBOL), GO terms (GO), KEGG pathway IDs (KEGG), ENSEMBL IDs (ENSEMBL) and chromosome start and stop locations (CHRLOC and CHRLOCEND). Each mapping will have a manual page that you can read to describe the data in the mapping and where it came from.

```
R> ?hgu95av2CHRLOC
```

Exercise 2
For the probes in 'probes' above, use the annotation mappings to find the chromosome start locations.

2.0.5 Manipulating Bimap Objects

Many filtering operations on the annotation *Bimap* objects require conversion of the *AnnDbBimap* into a *list*. In general, converting to lists will not be the most efficient way to filter the annotation data when using a SQLite-based package. Compare the following two examples for how you could get the 1st ten elements of the hgu95av2SYMBOL mapping. In the 1st case we have to get the entire mapping into list form, but in the second case we first subset the mapping object itself and this allows us to only convert the ten elements that we care about.

```
R> system.time(as.list(hgu95av2SYMBOL)[1:10])
R> ## vs:
R>
R> system.time(as.list(hgu95av2SYMBOL[1:10]))
```

There are many different kinds of *Bimap* objects in AnnotationDbi, but most of them are of class *AnnDbBimap*. All /RclassBimap objects represent data as a set of left and right keys. The typical usage of these mappings is to search for right keys that match a set of left keys that have been supplied by the user. But sometimes it is also convenient to go in the opposite direction.

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

```
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] "35870_at" "35761_at"
```

Note that 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 "ProbeAnnDbBimap")
```

```
R> identical(revx, revx2)
```

```
[1] TRUE
```

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

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

Note that most maps are reversible with `revmap`, but some (such as the more complex GO mappings), are not. Why is this? Because to reverse a mapping means that there has to be a "value" that will always become the "key" on the newly reversed map. And GO mappings have several distinct possibilities to choose from (GO ID, Evidence code or Ontology). In non-reversible cases like this, AnnotationDbi will usually provide a pre-defined reverse map. That way, you will always know what you are getting when you call `revmap`

While we are on the subject of GO and GO mappings, there are a series of special methods for GO mappings that can be called to find out details about these IDs. `Term`,`GOID`, `Ontology`, `Definition`,`Synonym`, and `Secondary` are all useful ways of getting additional information about a particular GO ID. For example:

```
R> Term("GO:0000018")
```

```
                       GO:0000018
"regulation of DNA recombination"
```

```
R> Definition("GO:0000018")
```

```


"Any process that modulates the frequency, rate or extent of DNA recombination, a pro
```

Exercise 3
Given the following set of RefSeq IDs: c("NG_005114","NG_007432","NG_008063"), Find the Entrez Gene IDs that would correspond to those. Then find the GO terms that are associated with those entrez gene IDs.
org.Hs.eg.db packages.

2.0.6 The Contents and Structure of Bimap Objects

Sometimes you may want to display or subset elements from an individual map. A *Bimap* interface is available to access the data in table (*data.frame*) format using [and toTable.

```
R> head(toTable(hgu95av2GO[probes]))
```

```
    probe_id      go_id Evidence Ontology
1    1702_at GO:0006915      TAS       BP
2    1702_at GO:0006955      TAS       BP
3    1702_at GO:0007166      TAS       BP
4    1702_at GO:0008283      TAS       BP
5   31882_at GO:0006364      TAS       BP
6 37033_s_at GO:0001659      IEA       BP
```

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 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 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. We refer to it as an 'undirected method':

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

```
  probe_id path_id
1 38187_at   00232
2 38187_at   00983
3 38187_at   01100
4 38912_at   00232
5 38912_at   00983
6 38912_at   01100
```

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

```
  probe_id path_id
1 38187_at   00232
```

```
2 38187_at   00983
3 38187_at   01100
4 38912_at   00232
5 38912_at   00983
6 38912_at   01100
```

Notice 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 38187_at IPI00644361 PF00797
2 38187_at IPI00930394 PF00797
3 38912_at IPI00004421 PF00797
4 38912_at IPI00789198 PF00797
5 33825_at IPI00550991 PF00079
6 33825_at IPI00607870 PF00079
```

```
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, hence we refer to these as 'directed methods':

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

```
[1] 12625
```

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

```
[1] 220
```

```
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] 220
```

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

You might be tempted to think that `Lkeys` and `Llength` will tell you all that you want to know about the left keys. But things are more complex than this, because not all keys are mapped. Often, you will only want to know about the keys that are mapped (ie. the ones that have a corresponding Rkey). To learn this you want to use the `mappedkeys` or the undirected variants `mappedLkeys` and `mappedRkeys`. Similarily, the `count.mappedkeys`, `count.mappedLkeys` and `count.mappedRkeys` methods are very fast ways to determine how many keys are mapped. Accessing keys like this is usually very fast and so it can be a decent strategy to subset the mapping by 1st using the mapped keys that you want to find.

```
R> x = hgu95av2ENTREZID[1:10]
R> ## Directed methods
R> mappedkeys(x)           # mapped keys

 [1] "1000_at"   "1001_at"   "1002_f_at" "1003_s_at" "1004_at"
 [6] "1005_at"   "1006_at"   "1007_s_at" "1008_f_at" "1009_at"

R> count.mappedkeys(x)     # nb of mapped keys

[1] 10

R> ## Undirected methods
R> mappedLkeys(x)          # mapped left keys

 [1] "1000_at"   "1001_at"   "1002_f_at" "1003_s_at" "1004_at"
 [6] "1005_at"   "1006_at"   "1007_s_at" "1008_f_at" "1009_at"
```

```
R> count.mappedLkeys(x)    # nb of mapped Lkeys
```

```
[1] 10
```

If you want to find keys that are not mapped to anything, you might want to use `isNA`.

```
R> y = hgu95av2ENTREZID[isNA(hgu95av2ENTREZID)]     # usage like is.na()
R> Lkeys(y)[1:4]
```

```
[1] "1047_s_at" "1089_i_at" "108_g_at"  "1090_f_at"
```

Exercise 4
How many probesets do not have a GO mapping for the hgu95av2.db package? How many have no mapping? Find a probeset that has a GO mapping. Now look at the GO mappings for this probeset in table form.

2.0.7 Some specific examples

Lets use what we have learned to get information about the probes that are are not assigned to a chromosome:

```
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"  "MT" "Un" "X"  "Y"
```

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

```
[1] "MT" "Un"
```

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

```
  probe_id chromosome
1 31593_at         Un
```

To get this in the classic named-list format:

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

```
[1] "MT" "Un"
```

```
R> z[["Y"]]
```

```
NULL
```

Many of the common methods for accessing *Bimap* objects return things in list format. This can be convenient. But you have to be careful about this if you want to use unlist(). For example the following will return multiple probes for each chromosome:

```
R> chrs = c("12","6")
R> mget(chrs, revmap(hgu95av2CHR[1:30]), ifnotfound=NA)
```

```
$`12`
[1] "1018_at"   "1019_g_at" "101_at"    "1021_at"

$`6`
[1] "1007_s_at" "1026_s_at" "1027_at"
```

But look what happens here if we try to unlist that:

```
R> unlist(mget(chrs, revmap(hgu95av2CHR[1:30]), ifnotfound=NA))
```

```
        121         122         123         124          61          62
  "1018_at" "1019_g_at"    "101_at"   "1021_at" "1007_s_at" "1026_s_at"
         63
  "1027_at"
```

Yuck! One trick that will sometimes help is to use Rfunctionunlist2. But be careful here too. Depending on what step comes next, Rfunctionunlist2 may not really help you...

```
R> unlist2(mget(chrs, revmap(hgu95av2CHR[1:30]), ifnotfound=NA))
```

```
         12          12          12          12           6           6
  "1018_at" "1019_g_at"    "101_at"   "1021_at" "1007_s_at" "1026_s_at"
          6
  "1027_at"
```

Lets ask if the probes in 'pbids' mapped to cytogenetic location "18q11.2"?

```
R> x <- hgu95av2MAP
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 for the reasons stated above:

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

2.0.8 Accessing probes that map to multiple targets

In many probe packages, some probes are known to map to multiple genes. The reasons for this can be biological as happens in the arabidopsis packages, but usually it is due to the fact that the genome builds that chip platforms were based on were less stable than desired. Thus what may have originally been a probe designed to measure one thing can end up measuring many things. Usually you don't want to use probes like this, because if they manufacturer doesn't know what they map to then their usefullness is definitely suspect. For this reason, by default all chip packages will normally hide such probes in the standard mappings. But sometimes you may want access to the answers that the manufacturer says such a probe will map to. In such cases, you will want to use the `toggleProbes` method. To use this method, just call it on a standard mapping and copy the result into a new mapping (you cannot alter the original mapping). Then treat the new mapping as you would any other mapping.

```
R>   ## How many probes?
R>   dim(hgu95av2ENTREZID)
```

```
[1] 11736     2
```

```
R>   ## Make a mapping with multiple probes exposed
R>   multi <- toggleProbes(hgu95av2ENTREZID, "all")
R>   ## How many probes?
R>   dim(multi)
```

```
[1] 12837     2
```

If you then decide that you want to make a mapping that has only multiple mappings or you wish to revert one of your maps back to the default state of only showing the single mappings then you can use toggleProbes to switch back and forth.

```
R>   ## Make a mapping with ONLY multiple probes exposed
R>   multiOnly <- toggleProbes(multi, "multiple")
R>   ## How many probes?
R>   dim(multiOnly)
```

```
[1] 1101    2
```

```
R>   ## Then make a mapping with ONLY single mapping probes
R>   singleOnly <- toggleProbes(multiOnly, "single")
R>   ## How many probes?
R>   dim(singleOnly)
```

```
[1] 11736     2
```

Finally, there are also a pair of test methods hasMultiProbes and hasSingleProbes that can be used to see what methods a mapping presently has exposed.

```
R>   ## Test the multiOnly mapping
R>   hasMultiProbes(multiOnly)
```

```
[1] TRUE
```

```
R>   hasSingleProbes(multiOnly)
```

```
[1] FALSE
```

```
R>   ## Test the singleOnly mapping
R>   hasMultiProbes(singleOnly)
```

```
[1] FALSE
```

```
R>   hasSingleProbes(singleOnly)
```

```
[1] TRUE
```

2.0.9 Using SQL to access things directly

While the mapping objects provide a lot of convenience, sometimes there are definite benefits to writing a simple SQL query. But in order to do this, it is necessary to know a few things. The 1st thing you will need to know is some SQL. Fortunately, it is quite easy to learn enough basic SQL to get stuff out of a database. Here are 4 basic SQL things that you may find handy:

First, you need to know about SELECT statements. A simple example would look something like this:

SELECT * FROM genes;

Which would select everything from the genes table.

SELECT gene_id FROM genes;

Will select only the gene_id field from the genes table.

Second you need to know about WHERE clauses:

SELECT gene_id,_id FROM genes WHERE gene_id=1;

Will only get records from the genes table where the gene_id is = 1.

Thirdly, you will want to know about an inner join:

SELECT * FROM genes,chromosomes WHERE genes._id=chromosomes._id;

This is only slightly more complicated to understand. Here we want to get all the records that are in both the 'genes' and 'chromosomes' tables, but we only want ones where the '_id' field is identical. This is known as an inner join because we only want the elements that are in both of these tables with respect to '_id'. There are other kinds of joins that are worth learning about, but most of the time, this is all you will need to do.

Finally, it is worthwhile to learn about the AS keyword which is useful for making long queries easier to read. For the previous example, we could have written it this way to save space:

SELECT * FROM genes AS g,chromosomes AS c WHERE g._id=c._id;

In a simple example like this you might not see a lot of savings from using AS, so lets consider what happens when we want to also specify which fields we want:

SELECT g.gene_id,c.chromosome FROM genes AS g,chromosomes AS c WHERE g._id=c._id;

Now you are most of the way there to being able to query the databases directly. The only other thing you need to know is a little bit about how to access these databases from R. With each package, you will also get a method that will print the schema for its database, you can view this to see what sorts of tables are present etc.

```
R> org.Hs.eg_dbschema()
```

To access the data in a database, you will need to connect to it. Fortunately, each package will automatically give you a connection object to that database when it loads.

```
R> org.Hs.eg_dbconn()
```

You can use this connection object like this:

```
R> query <- "SELECT gene_id FROM genes LIMIT 10;"
R> result = dbGetQuery(org.Hs.eg_dbconn(), query)
R> result
```

Exercise 5
Retrieve the entrez gene ID and chromosome by using a database query. Show how you could do the same thing by using `toTable`

2.0.10 Combining data from multiple annotation packages at the SQL level

For a more complex example, consider the task of obtaining all gene symbols 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> 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> 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 hgu95av2 database. In this case, this is a chip package, so you will also need to see the schema for the organism package that it depends on. To learn what package it depends on, look at the ORGPKG value:

```
R> hgu95av2ORGPKG
```

Then you can see that schema by looking at its schema method:

```
R> org.Hs.eg_dbschema()
```

So now we can see that we want to connect the data in the go_bp, and symbol tables from the org.Hs.eg.sqlite database along with the probes data in the hgu95av2.sqlite database. How can we do that?

It turns out that one of the great conveniences of SQLite is that it allows other databases to be 'ATTACHed'. Thus, we can keep our data in many differnt databases, and then 'ATTACH' them to each other in a modular fashion. The databases for a given build have been built together and frozen into a single version specifically to allow this sort of behavoir. To use this feature, the SQLite ATTACH command requires the filename for the database file on your filesystem. Fortunately, R provides a nice system independent way of getting that information. Note that the name of the database is always the same as the name of the package, with the suffix '.sqlite'.:

```
R> orgDBLoc = system.file("extdata", "org.Hs.eg.sqlite", package="org.Hs.eg.db")
R> attachSQL = paste("ATTACH '", orgDBLoc, "' AS orgDB;", sep = "")
R> dbGetQuery(hgu95av2_dbconn(), attachSQL)

NULL
```

Finally, you can assemble a cross-db sql query and use the helper function as follows. Note that when we want to refer to tables in the attached database, we have to use the 'orgDB' prefix that we specified in the 'ATTACH' query above.:

```
R> system.time({
 SQL <- "SELECT DISTINCT probe_id,symbol FROM probes, orgDB.gene_info AS gi, orgDB.ge
 zz <- dbGetQuery(hgu95av2_dbconn(), SQL)
 })

   user  system elapsed 
  0.168   0.004   0.171 

R> #its a good idea to always DETACH your database when you are finished...
R> dbGetQuery(hgu95av2_dbconn(), "DETACH orgDB"         )

NULL
```

Exercise 6
Retrieve the entrez gene ID, chromosome location information and cytoband infomration by using a single database query.

Exercise 7
Expand on the example in the text above to combine data from the hgu95av2.db and org.Hs.eg.db with the GO.db package so as to include the GO ID, and term definition in the output.

The version number of R and packages loaded for generating the vignette were:

```
R version 2.10.1 (2009-12-14)
x86_64-unknown-linux-gnu

locale:
 [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
 [3] LC_TIME=en_US.UTF-8        LC_COLLATE=en_US.UTF-8
 [5] LC_MONETARY=C              LC_MESSAGES=en_US.UTF-8
 [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
 [9] LC_ADDRESS=C               LC_TELEPHONE=C
[11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C

attached base packages:
[1] tools     stats     graphics  grDevices utils     datasets
[7] methods   base

other attached packages:
[1] GO.db_2.3.5          hgu95av2.db_2.3.5    org.Hs.eg.db_2.3.6
[4] RSQLite_0.8-4        DBI_0.2-5            AnnotationDbi_1.8.2
[7] Biobase_2.6.1
```