Creating select Interfaces for custom Annotation resources

Marc Carlson and Valerie Obenchain

January 15, 2016

1 Introduction

The most common interface for retrieving data in *Bioconductor* is now the `select` method. The interface provides a simple way of extracting data.

There are really 4 methods that work together to allow a `select` interface. The 1st one is `columns`, which tells you about what kinds of values you can retrieve as columns in the final result.

```
library(Homo.sapiens)

## Loading required package:  AnnotationDbi

## Loading required package:  stats4

## Loading required package:  BiocGenerics

## Loading required package:  parallel

##
## Attaching package:  'BiocGenerics'

## The following objects are masked from 'package:parallel':
##
##    clusterApply, clusterApplyLB, clusterCall, clusterEvalQ, clusterExport,
##    clusterMap, parApply, parCapply, parLapply, parLapplyLB, parRapply,
##    parSapply, parSapplyLB

## The following objects are masked from 'package:stats':
##
##    IQR, mad, xtabs

## The following objects are masked from 'package:base':
##
##    Filter, Find, Map, Position, Reduce, anyDuplicated, append, as.data.frame,
##    as.vector, cbind, colnames, do.call, duplicated, eval, evalq, get, grep,
##    grepl, intersect, is.unsorted, lapply, lengths, mapply, match, mget, order,
##    paste, pmax, pmax.int, pmin, pmin.int, rank, rbind, rownames, sapply,
##    setdiff, sort, table, tapply, union, unique, unlist, unsplit

## Loading required package:  Biobase
```

```
## Welcome to Bioconductor
##
##     Vignettes contain introductory material; view with 'browseVignettes()'.  To
##     cite Bioconductor, see 'citation("Biobase")', and for packages
##     'citation("pkgname")'.

## Loading required package:  IRanges

## Loading required package:  S4Vectors

## Loading required package:  OrganismDbi

## Loading required package:  GenomicFeatures

## Loading required package:  GenomeInfoDb

## Loading required package:  GenomicRanges

## Loading required package:  GO.db

## Loading required package:  DBI

##

## Loading required package:  org.Hs.eg.db

##

## Loading required package:  TxDb.Hsapiens.UCSC.hg19.knownGene

## Now getting the GODb Object directly

## Now getting the OrgDb Object directly

## Now getting the TxDb Object directly
```

```r
columns(Homo.sapiens)
```

```
##  [1] "ACCNUM"       "ALIAS"        "CDSCHROM"     "CDSEND"       "CDSID"
##  [6] "CDSNAME"      "CDSSTART"     "CDSSTRAND"    "DEFINITION"   "ENSEMBL"
## [11] "ENSEMBLPROT"  "ENSEMBLTRANS" "ENTREZID"     "ENZYME"       "EVIDENCE"
## [16] "EVIDENCEALL"  "EXONCHROM"    "EXONEND"      "EXONID"       "EXONNAME"
## [21] "EXONRANK"     "EXONSTART"    "EXONSTRAND"   "GENEID"       "GENENAME"
## [26] "GO"           "GOALL"        "GOID"         "IPI"          "MAP"
## [31] "OMIM"         "ONTOLOGY"     "ONTOLOGYALL"  "PATH"         "PFAM"
## [36] "PMID"         "PROSITE"      "REFSEQ"       "SYMBOL"       "TERM"
## [41] "TXCHROM"      "TXEND"        "TXID"         "TXNAME"       "TXSTART"
## [46] "TXSTRAND"     "TXTYPE"       "UCSCKG"       "UNIGENE"      "UNIPROT"
```

The next method is keytypes which tells you the kinds of things that can be used as keys.

```r
keytypes(Homo.sapiens)
```

```
##  [1] "ACCNUM"       "ALIAS"        "CDSID"        "CDSNAME"      "DEFINITION"
##  [6] "ENSEMBL"      "ENSEMBLPROT"  "ENSEMBLTRANS" "ENTREZID"     "ENZYME"
## [11] "EVIDENCE"     "EVIDENCEALL"  "EXONID"       "EXONNAME"     "GENEID"
## [16] "GENENAME"     "GO"           "GOALL"        "GOID"         "IPI"
```

```
## [21] "MAP"          "OMIM"        "ONTOLOGY"     "ONTOLOGYALL"  "PATH"
## [26] "PFAM"         "PMID"        "PROSITE"      "REFSEQ"       "SYMBOL"
## [31] "TERM"         "TXID"        "TXNAME"       "UCSCKG"       "UNIGENE"
## [36] "UNIPROT"
```

The third method is `keys` which is used to retrieve all the viable keys of a particular type.

```
k <- head(keys(Homo.sapiens,keytype="ENTREZID"))
k
```

```
## [1] "1"  "2"  "3"  "9"  "10" "11"
```

And finally there is `select`, which extracts data by using values supplied by the other method.

```
result <- select(Homo.sapiens, keys=k,
                 columns=c("TXNAME","TXSTART","TXSTRAND"),
                 keytype="ENTREZID")
```

```
## 'select()' returned 1:many mapping between keys and columns
```

```
head(result)
```

```
##   ENTREZID     TXNAME TXSTRAND  TXSTART
## 1        1 uc002qsd.4        - 58858172
## 2        1 uc002qsf.2        - 58859832
## 3        2 uc001qvk.1        -  9220304
## 4        2 uc009zgk.1        -  9220304
## 5        3 uc021qum.1        -  9381129
## 6        9 uc010ltd.3        + 18027971
```

But why would we want to implement these specific methods? It's a fair question. Why would we want to write a select interface for our annotation data? Why not just save a .rda file to the data directory and be done with it? There are basically two reasons for this. The 1st reason is convenience for end users. When your end users can access your data using the same four methods that they use everywhere else, they will have a more effortless time retrieving their data. And things that benefit your users benefit you.

The second reason is that by enabling a consistent interface across all annotation resources, we allow for things to be used in a programmatic manner. By implementing a select interface, we are creating a universal API for the whole project.

Lets look again at the example I described above and think about what is happening. The *Homo.sapiens* package is able to integrate data from many different resources largely because the separate resources all implemented a select method. This allows the *OrganismDbi* package to pull together resources from *org.Hs.eg.db*, *GO.db* and *TxDb.Hsapiens.UCSC.hg19.knownGene*.

If these packages all exposed different interfaces for retrieving the data, then it would be a lot more challenging to retrieve it, and writing general code that retrieved the appropriate data would be a lost cause. So implementing a set of select methods is a way to convert your package from a data file into an actual resource.

Figure 1: Packages and relationships represented by the Homo.sapiens package

2 Creating other kinds of Annotation packages

A few more automated options already exist for generating specific 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 *AnnotationForge* package. And, for users who seek to make a probe package, there is another vignette called *Creating probe packages* that is also in the *AnnotationForge* 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 web resource or database and you wanted to expose it to the world using something like this new `select` method interface? How could you go about this?

3   Retrieving data from a web resource

If you choose to expose a web resource, then you will need to learn some skills for retrieving that data from the web. The *R* programming language has tools that can help you interact with web resources, pulling down files that are tab delimited or formatted as XML etc. There are also packages that can help you parse what you retrieve. In this section we will describe some of these resources in the context of the uniprot web service, and give examples to demonstrate how you can expose resources like this for your own purposes.

These days many web services are exposed using a representational state transfer or RESTful interface. An example of this are the services offered at Uniprot. Starting with the uniprot base URI you can add details to simply indicate what it is that you wish to retrieve.

So in the case of Uniprot the base URI for the service we want today is this:

    http://www.uniprot.org/uniprot/

This URI can be extended to retrieve individual uniprot records by specifying a query argument like this:

    http://www.uniprot.org/uniprot/?query=P13368

We can then request multiple records like this:

    http://www.uniprot.org/uniprot/?query=P13368+or+Q6GZX4

And we can ask that the records be returned to us in tabular form by adding another argument like this.

    http://www.uniprot.org/uniprot/?query=P13368+or+Q6GZX4&format=tab

As you might guess, each RESTful interface is a little different, but you can easily see how once you read the documentation for a given RESTful interface, you can start to retrieve the data in *R*. Here is an example.

```
uri <- 'http://www.uniprot.org/uniprot/?query='
ids <- c('P13368', 'Q6GZX4')
idStr <- paste(ids, collapse="+or+")
format <- '&format=tab'
fullUri <- paste0(uri,idStr,format)
read.delim(fullUri)
```

```
##    Entry  Entry.name   Status                         Protein.names        Gene.names
## 1 Q6GZX4  001R_FRG3G reviewed Putative transcription factor 001R          FV3-001R
## 2 P13368 7LESS_DROME reviewed     Protein sevenless (EC 2.7.10.1) sev HD-265 CG18085
##                              Organism Length
## 1 Frog virus 3 (isolate Goorha) (FV-3)    256
## 2  Drosophila melanogaster (Fruit fly)   2554
```

Exercise 1
If you use the columns argument you can also specify which columns you want returned. So for example, you can choose to only have the sequence and id columns returned like this:

    http://www.uniprot.org/uniprot/?query=P13368+or+Q6GZX4&format=tab&columns=id,sequence

Use this detail about the uniprot web service along with what was learned above to write a function that takes a character vector of uniprot IDs and another character vector of columns arguments and then returns the appropriate values. Be careful to filter out any extra records that the service returns.

Solution:

```r
getUniprotGoodies  <- function(query, columns)
{
    ## query and columns start as a character vectors
    qstring <- paste(query, collapse="+or+")
    cstring <- paste(columns, collapse=",")
    uri <- 'http://www.uniprot.org/uniprot/?query='
    fullUri <- paste0(uri,qstring,'&format=tab&columns=',cstring)
    dat <- read.delim(fullUri, stringsAsFactors=FALSE)
    ## now remove things that were not in the specific original query...
    dat <- dat[dat[,1] %in% query,]
    dat
}
```

3.1   Parsing XML

Data for the previous example were downloaded from Uniprot in tab-delimited format. This is a convienent output to work with but unfortunately not always available. XML is still very common and it is useful to have some familiarity with parsing it. In this section we give a brief overview to using the *XML* package for navigating XML data.

The *XML* package provides functions to parse XML in both the tree-based DOM (document object model) or the event-driven SAX (Simple API for XML). We will use the DOM approach. The XML is first parsed into a tree-structure where the different elements of the data are nodes. The elements are processed by traversing the tree and generating a user-level representation of the nodes. XPath syntax is used to traverse the nodes. A detailed description of XPath can be found at http://www.w3.org/xml.

Retrieve the data: Data will be retrieved for the same id's as in the previous example. Unlike tab-delimited, the XML queries cannot be subset by column so the full record will be returned for each id. Details for what is possible with each type of data retrieval are found at http://www.uniprot.org/faq/28.

Parse the XML into a tree structure with `xmlTreeParse`. When `useInternalNodes=TRUE` and no `handlers` are specified the return value is a reference to C-level nodes. This storage mode allows us to traverse the tree of data in C instead of R objects.

```r
library(XML)
uri <- "http://www.uniprot.org/uniprot/?query=P13368+or+Q6GZX4&format=xml"
xml <- xmlTreeParse(uri, useInternalNodes=TRUE)
```

XML namespace: XML pages can have namespaces which facilatate the use of different XML vocabularies by resolving conflicts arrising from identical tags. Namepaces are represented by a uri pointing to an XML schema page. When a namespace is defined on a node in an XML document it must be included in the XPath expression.

Use the `xmlNamespaceDefinitions` function to check if the XML has a namespace.

```
defs <- xmlNamespaceDefinitions(xml, recurisve=TRUE)
defs

## [[1]]
## $id
## [1] ""
##
## $uri
## [1] "http://uniprot.org/uniprot"
##
## $local
## [1] TRUE
##
## attr(,"class")
## [1] "XMLNamespaceDefinition"
##
## $xsi
## $id
## [1] "xsi"
##
## $uri
## [1] "http://www.w3.org/2001/XMLSchema-instance"
##
## $local
## [1] TRUE
##
## attr(,"class")
## [1] "XMLNamespaceDefinition"
##
## attr(,"class")
## [1] "XMLNamespaceDefinitions"
```

The presence of uri's confirm there is a namespace. Alternatively we could have looked at the XML nodes for declarstions of the form `xmlns:myNamespace="http://www.namspace.org"`. We organize the namespaces and will use them directly in parsing.

```
ns <- structure(sapply(defs, function(x) x$uri), names=names(defs))
```

Parsing with XPath: There are two high level 'entry' nodes which represent the two id's requested in the original query.

```
entry <- getNodeSet(xml, "//ns:entry", "ns")
xmlSize(entry)

## [1] 2
```

To get an idea of the data structure we first list the attributes of the top nodes and extract the names.

```
nms <- xpathSApply(xml, "//ns:entry/ns:name", xmlValue, namespaces="ns")
attrs <- xpathApply(xml, "//ns:entry", xmlAttrs, namespaces="ns")
names(attrs) <- nms
attrs

## $`001R_FRG3G`
##      dataset      created     modified      version
## "Swiss-Prot" "2011-06-28" "2015-04-01"         "30"
##
## $`7LESS_DROME`
##      dataset      created     modified      version
## "Swiss-Prot" "1990-01-01" "2015-11-11"        "155"
```

Next, inspect the direct children of each node.

```
fun1 <- function(elt) unique(names(xmlChildren(elt)))
xpathApply(xml, "//ns:entry", fun1, namespaces="ns")

## [[1]]
##  [1] "accession"       "name"            "protein"         "gene"
##  [5] "organism"        "organismHost"    "reference"       "comment"
##  [9] "dbReference"     "proteinExistence" "keyword"         "feature"
## [13] "evidence"        "sequence"
##
## [[2]]
##  [1] "accession"       "name"            "protein"         "gene"
##  [5] "organism"        "reference"       "comment"         "dbReference"
##  [9] "proteinExistence" "keyword"         "feature"         "evidence"
## [13] "sequence"
```

Query Q6GZX4 has 2 'feature' nodes and query P13368 has 48.

```
Q6GZX4 <- "//ns:entry[ns:accession='Q6GZX4']/ns:feature"
xmlSize(getNodeSet(xml, Q6GZX4, namespaces="ns"))

## [1] 2

P13368 <- "//ns:entry[ns:accession='P13368']/ns:feature"
xmlSize(getNodeSet(xml, P13368, namespaces="ns"))

## [1] 48
```

List all possible values for the 'type' attribute of the 'feature' nodes.

```
path <- "//ns:feature"
unique(xpathSApply(xml, path, xmlGetAttr, "type", namespaces="ns"))

##  [1] "chain"                             "compositionally biased region"
##  [3] "topological domain"                "transmembrane region"
##  [5] "domain"                            "repeat"
##  [7] "nucleotide phosphate-binding region" "active site"
##  [9] "binding site"                      "modified residue"
## [11] "glycosylation site"                "mutagenesis site"
```

```
## [13] "sequence conflict"
```

XPath allows the construction of complex queries to pull out specific subsets of data. Here we extract the features with 'type=sequence conflict' for query P13368.

```
path <- "//ns:entry[ns:accession='P13368']/ns:feature[@type='sequence conflict']"
data.frame(t(xpathSApply(xml, path, xmlAttrs, namespaces="ns")))

##                    type                 description ref evidence
## 1 sequence conflict        In Ref. 1; AAA28882.   1        8
## 2 sequence conflict        In Ref. 3; AAF47992.   3        8
## 3 sequence conflict        In Ref. 3; AAF47992.   3        8
## 4 sequence conflict        In Ref. 3; AAF47992.   3        8
## 5 sequence conflict        In Ref. 3; AAF47992.   3        8
## 6 sequence conflict In Ref. 2; CAA31960/CAB55310.   2        8
## 7 sequence conflict        In Ref. 1; AAA28882.   1        8
```

Put the sequences in an AAStringSet and add the names.

```
library(Biostrings)

## Loading required package:  XVector

path <- "//ns:entry/ns:sequence"
seqs <- xpathSApply(xml, path, xmlValue, namespaces="ns")
aa <- AAStringSet(unlist(lapply(seqs, function(elt) gsub("\n", "", elt)),
    use.names=FALSE))
names(aa) <- nms
aa

##   A AAStringSet instance of length 2
##     width seq                                                         names
## [1]   256 MAFSAEDVLKEYDRRRRMEALLLSLYYP...VKTGSKGVLYDDSFRKIYTDLGWKFTPL 001R_FRG3G
## [2]  2554 MTMFWQQNVDHQSDEQDKQAKGAAPTKR...RPEDENLTLREVPLKDKQLYANEGVSRL 7LESS_DROME
```

4   Setting up a package to expose a web service

In order to expose a web service using select, you will need to create an object that will be loaded at the time when the package is loaded. Unlike with a database, the purpose of this object is pretty much purely for dispatch. We just need select and it's friends to know which select method to call

The first step is to create an object. Creating an object is simple enough:

```
setClass("uniprot", representation(name="character"),
         prototype(name="uniprot"))
```

Once you have a class defined, all you need is to make an instance of this class. Making an instance is easy enough:

```
    uniprot <- new("uniprot")
```

But of course it's a little more complicated because one of these objects will need to be spawned up whenever our package loads. This is acclomplished by calling the `.onLoad` function in the zzz.R file. The following code will create an object, and then assign it to the package namespace as the package loads.

```r
.onLoad <- function(libname, pkgname)
{
    ns <- asNamespace(pkgname)
    uniprot <- new("uniprot")
    assign("uniprot", uniprot, envir=ns)
    namespaceExport(ns, "uniprot")
}
```

5 Creating package accessors for a web service

At this point you have all that you need to know in order to implement `keytype`,`columns`,`keys` and `select` for your package. In this section we will explore how you could implement some of these if you were making a package that exposed uniprot.

5.1 Example: creating `keytypes` and `columns` methods

The `keytype` and `columns` methods are always the 1st ones you should implement. They are the easiest, and their existence is required to be able to use `keys` or `select`. In this simple case we only have one value that can be used as a keytype, and that is a UNIPROT ID.

```r
setMethod("keytypes", "uniprot",function(x){return("UNIPROT")})

## [1] "keytypes"

uniprot <- new("uniprot")
keytypes(uniprot)

## [1] "UNIPROT"
```

So what about columns? Well it's not a whole lot more complicated in this case since we are limited to things that we can return from the web service. Since this is just an example, lets limit it to the following fields: "ID", "SEQUENCE", "ORGANISM".

```r
setMethod("columns", "uniprot",
          function(x){return(c("ID", "SEQUENCE", "ORGANISM"))})

## [1] "columns"

columns(uniprot)

## [1] "ID"       "ORGANISM" "SEQUENCE"
```

Also, notice how for both `keytypes` and `columns` I am using all capital letters. This is a style adopted throughout the project.

5.2   Example 2: creating a `select` method

At this point we have enough to be able to make a select method.

Exercise 2
Using what you have learned above, and the helper function from earlier, define a select method. This select method will have a default `keytype` of "UNIPROT".

Solution:

```
.select <- function(x, keys, columns){
    colsTranslate <- c(id='ID', sequence='SEQUENCE', organism='ORGANISM')
    columns <- names(colsTranslate)[colsTranslate %in% columns]
    getUniprotGoodies(query=keys, columns=columns)
}
setMethod("select", "uniprot",
    function(x, keys, columns, keytype)
{
    .select(keys=keys, columns=columns)
})

## [1] "select"

select(uniprot, keys=c("P13368","P20806"), columns=c("ID","ORGANISM"))
```

6   Retrieving data from a database resource

If your package is retrieving data from a database, then there are some additional skills you will need to be able to interface with this database from *R*. This section will introduce you to those skills.

6.1   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:

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

But in our case the connection has already been created here as part of the object that was generated when the package was loaded:

```
require(hom.Hs.inp.db)
```

```
## Loading required package:  hom.Hs.inp.db

##

str(hom.Hs.inp.db)

## Reference class 'InparanoidDb' [package "AnnotationDbi"] with 2 fields
##  $ conn        :Formal class 'SQLiteConnection' [package "RSQLite"] with 5 slots
##   .. ..@ Id                 :<externalptr>
##   .. ..@ dbname             : chr "/home/biocbuild/bbs-3.2-bioc/R/library/hom.Hs.inp.db/extdat
##   .. ..@ loadable.extensions: logi TRUE
##   .. ..@ flags              : int 1
##   .. ..@ vfs                : chr ""
##  $ packageName: chr "hom.Hs.inp.db"
##  and 14 methods.
```

So we can do something like below:

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

## <SQLiteConnection>

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

## <SQLiteConnection>

## or we can just call the provided convenience function
## from when this package loads:
hom.Hs.inp_dbconn()

## <SQLiteConnection>
```

6.2   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.

```
con <- AnnotationDbi::dbconn(hom.Hs.inp.db)
head(dbListTables(con))

## [1] "Acyrthosiphon_pisum"   "Aedes_aegypti"          "Anopheles_gambiae"
## [4] "Apis_mellifera"        "Arabidopsis_thaliana"  "Aspergillus_fumigatus"

dbListFields(con, "Mus_musculus")

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

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.

```
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
```

6.3   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:

Table sna

foo	bar
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:

Table sna			Table fu	
foo	bar		foo	bo
1	baz		1	hi
2	boo		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.

6.4   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:

```
head(dbListTables(con))
```

```
## [1] "Acyrthosiphon_pisum"   "Aedes_aegypti"         "Anopheles_gambiae"
## [4] "Apis_mellifera"        "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`

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

```
## [1] "inp_id"     "clust_id"     "species"     "score"       "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`:

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

7   Setting up a package to expose a SQLite database object

For the sake of simplicity, lets 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 *Annotationsible.ddb* 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*.

```
.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 *hom.Hs.inp.db* package.

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

When the code above is run (at load time) the name of the package (AKA "pkgname", which is a parameter that will be passed into `.onLoad`) is then 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 it will be loaded for the user.

8   Creating package accessors for databases

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: `columns`,`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 *TxDb* objects) may also be warranted.

8.1   Examples: creating a `columns` and `keytypes` method

Now lets suppose that we want to define a `columns` 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?

```
.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
        list <- toupper(list)
        dbDisconnect(con)
        list
}


## Then make this into a method
setMethod("columns", "InparanoidDb", .cols(x))
## Then we can call it
columns(hom.Hs.inp.db)
```

Notice again 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 out 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 columns (in this case). What if you needed to translate back to the lowercase table names? Also write an quick helper function to do that.

Solution:
```
setMethod("keytypes", "InparanoidDb", .cols(x))
## Then we can call it
keytypes(hom.Hs.inp.db)

## refactor of .cols
.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]
    dbDisconnect(con)
    list
}
.cols <- function(x)
{
    list <- .getLCcolnames(x)
    ## Then just to format things in the usual way
    toupper(list)
}
## Test:
columns(hom.Hs.inp.db)
```

```
## new helper function:
.getTableNames <- function(x)
{
    LC <- .getLCcolnames(x)
    UC <- .cols(x)
    names(UC) <- LC
    UC
}
.getTableNames(hom.Hs.inp.db)
```

8.2 Example: creating a `keys` method

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:

```
.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)
    res <- as.vector(t(res))
    dbDisconnect(con)
    res
}

setMethod("keys", "InparanoidDb", .keys(x, keytype))
## Then we can call it
keys(hom.Hs.inp.db, "TRICHOPLAX_ADHAERENS")
```

9 Creating a database resource from available data

Sometimes you may have a lot of data that you want to organize into a database. Or you may have another existing database that you wish to convert into a SQLite database. This section will deal with some simple things you can do to create and import a SQLite database of your own.

9.1   Making a new connection

First, lets close the connection to our other DB:

```
    dbDisconnect(con)
```

```
## [1] TRUE
```

Then lets make a new database. Notice that we specify the database name with "dbname" This allows it to be written to disc instead of just memory.

```
drv <- dbDriver("SQLite")
dbname <- file.path(tempdir(), "myNewDb.sqlite")
con <- dbConnect(drv, dbname=dbname)
```

9.2   Importing data

Imagine that we want to reate a database and then put a table in it called genePheno to store the genes mutated and a phenotypes associated with each. Plan for genePheno to hold the following gene IDs and phenotypes (as a toy example):

```
data = data.frame(id=c(1,2,9),
                  string=c("Blue",
                           "Red",
                           "Green"),
                  stringsAsFactors=FALSE)
```

Making the table is very simple, and just involves a create table statement.

```
    CREATE Table genePheno (id INTEGER, string TEXT);
```

The SQL create table statement just indicates what the table is to be called, as well as the different fields that will be present and the type of data each field is expected to contain.

```
dbGetQuery(con, "CREATE Table genePheno (id INTEGER, string TEXT)")
```

But putting the data into the database is a little bit more delicate. We want to take control over which columns we want to insert from our `data.frame`. Fortunately, the RSQLite package provides these facilities for us.

```
names(data) <- c("id","string")
sql <- "INSERT INTO genePheno VALUES ($id, $string)"
dbBegin(con)
```

```
## [1] TRUE
```

```
dbGetPreparedQuery(con, sql, bind.data = data)
dbCommit(con)
```

```
## [1] TRUE
```

Please notice that we want to use strings instead of factors in our data.frame. If you insert the data as factors, you may not be happy with what ends up in the DB.

9.3   Attaching other database resources

In SQLite it is possible to attach another database to your session and then query across both resources as if they were the same DB.

The SQL what we want looks quite simple:

```
ATTACH "TxDb.Hsapiens.UCSC.hg19.knownGene.sqlite" AS db;
```

So in R we need to do something similar to this:

```
db <- system.file("extdata", "TxDb.Hsapiens.UCSC.hg19.knownGene.sqlite",
                  package="TxDb.Hsapiens.UCSC.hg19.knownGene")
dbGetQuery(con, sprintf("ATTACH '%s' AS db",db))
```

Here we have attached a DB from one of the packages that this vignette required you to have installed, but we could have attached any SQLite database that we provided a path to.

Once we have attached the database, we can join to it's tables as if they were in our own database. All that is required is a prefix, and some knowledge about how to do joins in SQL. In the end the SQL to take advantage of the attached database looks like this:

```
SELECT * FROM db.gene AS dbg, genePheno AS gp
WHERE dbg.gene_id=gp.id;
```

Then in R:

```
  sql <- "SELECT * FROM db.gene AS dbg,
          genePheno AS gp WHERE dbg.gene_id=gp.id"
  res <- dbGetQuery(con, sql)
  res
```

```
##    gene_id _tx_id id string
## 1        1  70455  1   Blue
## 2        1  70456  1   Blue
## 3        2  47665  2    Red
## 4        2  47666  2    Red
## 5        9  31934  9  Green
## 6        9  31938  9  Green
## 7        9  31939  9  Green
## 8        9  31940  9  Green
## 9        9  31941  9  Green
## 10       9  31937  9  Green
## 11       9  31936  9  Green
## 12       9  31935  9  Green
## 13       9  31942  9  Green
```

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

```
## R version 3.2.3 (2015-12-10)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Ubuntu 14.04.3 LTS
##
```

```
## locale:
##  [1] LC_CTYPE=en_US.UTF-8        LC_NUMERIC=C               LC_TIME=en_US.UTF-8
##  [4] LC_COLLATE=C               LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
##  [7] LC_PAPER=en_US.UTF-8       LC_NAME=C                  LC_ADDRESS=C
## [10] LC_TELEPHONE=C             LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C
##
## attached base packages:
## [1] parallel  stats4    stats     graphics  grDevices utils     datasets  methods
## [9] base
##
## other attached packages:
##  [1] hom.Hs.inp.db_3.1.2                   Biostrings_2.38.3
##  [3] XVector_0.10.0                        XML_3.98-1.3
##  [5] Homo.sapiens_1.3.1                    TxDb.Hsapiens.UCSC.hg19.knownGene_3.2.2
##  [7] org.Hs.eg.db_3.2.3                    GO.db_3.2.2
##  [9] RSQLite_1.0.0                         DBI_0.3.1
## [11] OrganismDbi_1.12.1                    GenomicFeatures_1.22.8
## [13] GenomicRanges_1.22.3                  GenomeInfoDb_1.6.1
## [15] AnnotationDbi_1.32.3                  IRanges_2.4.6
## [17] S4Vectors_0.8.9                       Biobase_2.30.0
## [19] BiocGenerics_0.16.1                   knitr_1.12
##
## loaded via a namespace (and not attached):
##  [1] graph_1.48.0              magrittr_1.5             GenomicAlignments_1.6.3
##  [4] zlibbioc_1.16.0          BiocParallel_1.4.3       stringr_1.0.0
##  [7] highr_0.5.1              tools_3.2.3              SummarizedExperiment_1.0.2
## [10] lambda.r_1.1.7           futile.logger_1.4.1      RBGL_1.46.0
## [13] rtracklayer_1.30.1       formatR_1.2.1            futile.options_1.0.0
## [16] bitops_1.0-6             RCurl_1.95-4.7           biomaRt_2.26.1
## [19] evaluate_0.8             stringi_1.0-1            BiocInstaller_1.20.1
## [22] Rsamtools_1.22.0         BiocStyle_1.8.0
```