poplite vignette

Daniel Bottomly and Beth Wilmot

May 13, 2019

1 Introduction

Prior to utilizing a given database in a research context, it has to be designed, scripts written to format existing data to fit into the tables, and the data has to be loaded. The loading can be as simple as just inserting data into a given table or in addition it may need to respect columns from other tables. The `poplite` package was developed originally to simplify the process of population and management of SQLite databases using R. It provides a schema object and corresponding methods which allows a user to easily understand the database structure as well as extend or modify it. The database can be populated using this schema and the user can specify both raw data and a transformational function to apply prior to loading. This functionality facilitates loading of large, non-`data.frame` type objects as is demonstrated below in the 'VCF Database' section. It has since also incorportated and extended functionality from the `dplyr` package [Wickham and Francois, 2014] to provide a convienient query interface using `dplyr`'s verbs. Notably, cross table queries can be carried out automatically using the specified schema object.

2 Sample Tracking Database

We will start by working through a simple example that illustrates how to use `poplite` for many common database tasks. Here we will create a sample tracking database for DNA specimens collected on patients in a clinical research setting. Our example is a set of 3 `data.frames` which consists of randomly generated values resembling commonly collected data. We first load the package and the example data, going through each `data.frame` in turn.

2.1 Database Population

```
> library(poplite)
> data(clinical)
> ls()
```

```
[1] "clinical" "dna"      "samples"
```

The clinical `data.frame` contains information on a group of patients including sex, age, disease status as well as other variables/covariates.

```
> head(clinical)
```

```
  sample_id sex age status  var_wave_1 var_wave_2
1         1   M  19      1 -0.05198191 -0.6451140
2         2   F  18      0 -1.75323736  0.1653210
3         3   F  17      0  0.09932759  0.4388187
4         4   M   6      1 -0.57185006  0.8833028
```

```
5         5   F  10      0 -0.97400958 -2.0523370
6         6   M   8      1 -0.17990623 -1.6363793

>
```

The samples `data.frame` records whether a given patient, keyed by sample_id was observed in one of several 'waves' and whether they contributed a DNA sample.

```
> head(samples)

  sample_id wave did_collect
1         1    1           Y
2         1    2           Y
3         2    1           Y
4         2    2           N
5         3    1           Y
6         3    2           N

>
```

The dna `data.frame` provides some information (concentration in nanograms/microliter) on the DNA specimen collected from a given patient during a given wave.

```
> head(dna)

   sample_id wave  lab_id      ng.ul
1          1    1 dna_1_1   5.777775
2          1    2 dna_1_2 203.292147
3          2    1 dna_2_1 158.137716
5          3    1 dna_3_1 167.162274
9          5    1 dna_5_1 183.204669
10         5    2 dna_5_2  96.772302

>
```

As these data are already in table form, loading them should be relatively straightforward. The first step is the creation of a schema object of class `TableSchemaList` whose name is descriptive of the data therein. For `data.frames` a convenience function (`makeSchemaFromData`) is provided which does some basic checks and creates an appropriate table based on the column names and inferred types.

```
> sample.tracking <- makeSchemaFromData(clinical, "clinical")
> sample.tbsl <- makeSchemaFromData(samples, "samples")
> sample.tracking <- append(sample.tracking,  sample.tbsl)
> try(dna.tbsl <- makeSchemaFromData(dna, "dna"))
>
```

The error caused by attempting to create a schema object from the 'dna' `data.frame`, is due to the formatting of the column name of 'ng.ul' which is a valid R name, but not a valid SQLite name. A convienient way to fix this is to use the `correct.df.names` function as below. Note that this modified `data.frame` will be the one that has to be provided to the `populate` method.

```
> new.dna <- correct.df.names(dna)
> dna.tbsl <- makeSchemaFromData(new.dna, "dna")
> sample.tracking <- append(sample.tracking, dna.tbsl)
>
```

At this point, the database can be created and populated using the schema defined in 'sample.tracking' and the data present in the respective `data.frames`. However, this requires anyone who wants to query across the tables be able to determine the relationships between the tables. In some cases this can be determined by column names alone, in other cases it may not be so clear. The `poplite` package allows the specification of relationships between tables using the R formula syntax. This functionality is probably best illustrated by example.

The 'clinical' table is the starting point of this database as both the 'samples' and 'dna' tables refer back to it. In the case of both the 'samples' and 'dna' tables, the 'clinical' table can be refered to using the 'sample_id' column. In addition, the 'dna' table can be referred to by the 'samples' table using a combination of 'sample_id' and 'wave' and vice versa. These types of relationship can be created by the following:

```
> relationship(sample.tracking, from="clinical", to="samples") <- sample_id~sample_id
> relationship(sample.tracking, from="clinical", to="dna") <-sample_id~sample_id
> relationship(sample.tracking, from="samples", to="dna") <- sample_id+wave~sample_id+wave
```

Now that our schema is complete, we are now ready to populate our database. First, we create a `Database` object which simply contains both the schema as well as the file path to where the database resides. We then call the `populate` method and provide it with our named `data.frames`.

```
> sample.tracking.db <- Database(sample.tracking, tempfile())
> populate(sample.tracking.db, dna=new.dna, samples=samples, clinical=clinical)
>
```

2.2 Querying

As mentioned in the introduction, the query interface utilizes the approach of the `dplyr` package, whereby a small set of verbs are used to perform common queries. In addition to utilizing the single table verbs defined in `dplyr`, `poplite` defines multi-table versions of the `select` and `filter` verbs. The `select` verb allows the user to select columns from a table-like object, in this case an SQLite database. As the `poplite` version of `select` and `filter` can be used for any of the tables in the defined schema, the most important requirement is for the user to make sure the column(s) are unambigous in terms of the tables. This can be done in several ways:

(1) Only for the `select` statement, the `.tables` argument allows selecting all or part of the columns of a given table(s). If multiple tables are provided, then they are first joined (an 'inner join' in SQL terminology) using the specified relationships in the schema. This provides a convienent way to retrieve a combined version of the data in the database.

```
> select(sample.tracking.db, .tables="dna")

# Source:   table<dna> [?? x 5]
# Database: sqlite 3.22.0 [/tmp/Rtmpai18QC/file433b10d331]
   dna_ind sample_id  wave lab_id    ng_ul
     <int>     <int> <int> <chr>     <dbl>
 1       1         1     1 dna_1_1    5.78
 2       2         1     2 dna_1_2  203.
 3       3         2     1 dna_2_1  158.
 4       4         3     1 dna_3_1  167.
 5       5         5     1 dna_5_1  183.
 6       6         5     2 dna_5_2   96.8
 7       7         7     2 dna_7_2  113.
 8       8         8     2 dna_8_2   46.2
 9       9         9     2 dna_9_2   41.0
10      10        10     1 dna_10_1 176.
# ... with more rows

> select(sample.tracking.db, sample_id:lab_id, .tables="dna")

# Source:   lazy query [?? x 3]
# Database: sqlite 3.22.0 [/tmp/Rtmpai18QC/file433b10d331]
   sample_id  wave lab_id
```

```
       <int> <int> <chr>
 1         1     1 dna_1_1
 2         1     2 dna_1_2
 3         2     1 dna_2_1
 4         3     1 dna_3_1
 5         5     1 dna_5_1
 6         5     2 dna_5_2
 7         7     2 dna_7_2
 8         8     2 dna_8_2
 9         9     2 dna_9_2
10        10     1 dna_10_1
# ... with more rows
```

```
> select(sample.tracking.db, .tables=c("clinical","dna"))
```

```
# Source:   lazy query [?? x 11]
# Database: sqlite 3.22.0 [/tmp/Rtmpai18QC/file433b10d331]
   clinical_ind sample_id sex     age status var_wave_1 var_wave_2 dna_ind  wave
          <int>     <int> <chr> <int>  <int>      <dbl>      <dbl>   <int> <int>
 1            1         1 M        19      1    -0.0520     -0.645       1     1
 2            1         1 M        19      1    -0.0520     -0.645       2     2
 3            2         2 F        18      0    -1.75        0.165       3     1
 4            3         3 F        17      0     0.0993      0.439       4     1
 5            5         5 F        10      0    -0.974      -2.05        5     1
 6            5         5 F        10      0    -0.974      -2.05        6     2
 7            7         8 M        13      1    -1.99        1.05        8     2
 8            8        10 M         6      0     0.117       0.715      10     1
 9            8        10 M         6      0     0.117       0.715      11     2
10            9        11 M        20      1    -0.893       0.917      12     1
# ... with more rows, and 2 more variables: lab_id <chr>, ng_ul <dbl>
```

```
>
```

(2) If a column or set of columns uniquely identifies a table, then no further information is needed to execute the query.

```
> select(sample.tracking.db, sample_id:lab_id)
```

```
# Source:   lazy query [?? x 3]
# Database: sqlite 3.22.0 [/tmp/Rtmpai18QC/file433b10d331]
   sample_id  wave lab_id
       <int> <int> <chr>
 1         1     1 dna_1_1
 2         1     2 dna_1_2
 3         2     1 dna_2_1
 4         3     1 dna_3_1
 5         5     1 dna_5_1
 6         5     2 dna_5_2
 7         7     2 dna_7_2
 8         8     2 dna_8_2
 9         9     2 dna_9_2
10        10     1 dna_10_1
# ... with more rows
```

```
> head(filter(sample.tracking.db, sex == "M" & var_wave_1 > 0))
```

```
# Source:   lazy query [?? x 7]
# Database: sqlite 3.22.0 [/tmp/Rtmpai18QC/file433b10d331]
```

```
  clinical_ind sample_id sex     age status var_wave_1 var_wave_2
         <int>     <int> <chr> <int>  <int>      <dbl>      <dbl>
1            8        10 M         6      0      0.117      0.715
2           10        12 M        20      1      0.334     -2.66 
3           14        19 M        18      0      0.274      0.519
4           18        23 M        11      0      0.378      0.741
5           25        30 M        19      0      0.825     -0.239
6           31        38 M        17      0      0.190      0.314
```

```
> filter(sample.tracking.db, sample_id == 97 & var_wave_1 > 0)
```

```
# Source:   lazy query [?? x 7]
# Database: sqlite 3.22.0 [/tmp/Rtmpai18QC/file433b10d331]
# ... with 7 variables: clinical_ind <int>, sample_id <int>, sex <chr>,
#   age <int>, status <int>, var_wave_1 <dbl>, var_wave_2 <dbl>
```

```
>
```

Whereas the following fails because 'sample_id' is defined in several tables.

```
> try(filter(sample.tracking.db, sample_id == 97))
>
```

(3) A table can be specified in the query using a '.' similar to how it is done in SQL, i.e. tableX.ColumnY. This only should be done once per statement for `select` statments, but per variable for filter statements as is shown below. This restriction reflects how the queries are carried out. Each grouping of statements statements are applied to the specified or inferred table prior to joining. Note that for the `poplite filter` verb cross-table 'OR' statements are currently not supported.

```
> select(sample.tracking.db, dna.sample_id)
```

```
# Source:   lazy query [?? x 1]
# Database: sqlite 3.22.0 [/tmp/Rtmpai18QC/file433b10d331]
   sample_id
       <int>
 1         1
 2         1
 3         2
 4         3
 5         5
 6         5
 7         7
 8         8
 9         9
10        10
# ... with more rows
```

```
> select(sample.tracking.db, dna.sample_id:lab_id)
```

```
# Source:   lazy query [?? x 3]
# Database: sqlite 3.22.0 [/tmp/Rtmpai18QC/file433b10d331]
   sample_id  wave lab_id
       <int> <int> <chr>
 1         1     1 dna_1_1
 2         1     2 dna_1_2
 3         2     1 dna_2_1
 4         3     1 dna_3_1
 5         5     1 dna_5_1
 6         5     2 dna_5_2
 7         7     2 dna_7_2
 8         8     2 dna_8_2
 9         9     2 dna_9_2
10        10     1 dna_10_1
# ... with more rows
```

```
> filter(sample.tracking.db, clinical.sample_id == 97)
```

```
# Source:   lazy query [?? x 7]
# Database: sqlite 3.22.0 [/tmp/Rtmpai18QC/file433b10d331]
  clinical_ind sample_id sex     age status var_wave_1 var_wave_2
         <int>     <int> <chr> <int>  <int>      <dbl>      <dbl>
1           86        97 F         7      0      -1.24     -0.273
```

```
> filter(sample.tracking.db, clinical.status == 1 & dna.wave==2)
```

```
# Source:   lazy query [?? x 11]
# Database: sqlite 3.22.0 [/tmp/Rtmpai18QC/file433b10d331]
   clinical_ind sample_id sex     age status var_wave_1 var_wave_2 dna_ind  wave
          <int>     <int> <chr> <int>  <int>      <dbl>      <dbl>   <int> <int>
 1            1         1 M        19      1    -0.0520    -0.645        2     2
 2            7         8 M        13      1    -1.99       1.05         8     2
 3           10        12 M        20      1     0.334     -2.66        14     2
 4           11        13 F         9      1     0.411      1.11        16     2
 5           17        22 M        10      1    -0.210      1.21        25     2
 6           20        25 F        12      1     0.857      0.0652      30     2
 7           22        27 F        19      1     2.42       1.98        33     2
 8           24        29 F        13      1    -0.464     -1.32        35     2
 9           26        32 M        16      1    -0.589      0.152       38     2
10           28        35 M        18      1    -0.0143     0.373       43     2
# ... with more rows, and 2 more variables: lab_id <chr>, ng_ul <dbl>
```

```
>
```

The `poplite` query interface is 'opt-in', meaning that more complex queries can always be carried out directly using the methodology provided in the `dplyr` package or plain SQL statements after first connecting to the database file using RSQLite. Below are three approaches of performing the same query:

```
> #poplite + dplyr
> wave.1.samp.pop <- filter(select(sample.tracking.db, .tables=c("samples", "dna")), wave == 1)
> #dplyr
> src.db <- src_sqlite(dbFile(sample.tracking.db), create = F)
> samp.tab <- tbl(src.db, "samples")
> dna.tab <- tbl(src.db, "dna")
> wave.1.samp.dplyr <- inner_join(filter(samp.tab, wave == 1), dna.tab,
+     by=c("sample_id", "wave"))
> library(RSQLite)
> #RSQLite
> samp.db <- dbConnect(SQLite(), dbFile(sample.tracking.db))
> wave.1.samp.sql <- dbGetQuery(samp.db, 'SELECT * FROM samples JOIN dna
+     USING (sample_id, wave) WHERE wave == 1')
> dbDisconnect(samp.db)
> all.equal(as.data.frame(wave.1.samp.pop), wave.1.samp.sql)
```

```
[1] TRUE
```

```
> all.equal(as.data.frame(wave.1.samp.dplyr), wave.1.samp.sql)
```

```
[1] TRUE
```

```
>
```

2.3 Additional Features

Instead of directly cross referencing columns as we did for the database above in our relationships (e.g. sample_id - sample_id) we can also specify that we instead want to use the 'primary key' of a given table instead of one or more columns. Where the primary key is a column of integer values that uniquely identifies each row and by default it was added automatically by `makeSchemaFromData`. This approach is commonly used for lookup tables, for instance say we wanted a seperate table for gender and to only maintain an integer value specifying whether a patient was male, female or unknown for the clinical table. We could do this as follows where '.' is a shortcut that indicates that the primary key for the table should be used:

```
> gender <- data.frame(sex=unique(clinical$sex), stringsAsFactors=F)
> gend.tbsl <- makeSchemaFromData(gender, "gender")
> sample.tracking <- append(gend.tbsl, sample.tracking)
> relationship(sample.tracking, from="gender", to="clinical") <- .~sex
> sample.tracking.db <- Database(sample.tracking, tempfile())
> populate(sample.tracking.db, dna=new.dna, samples=samples, clinical=clinical, gender=gender)
> select(sample.tracking.db, .tables="gender")
```

```
# Source:   table<gender> [?? x 2]
# Database: sqlite 3.22.0 [/tmp/Rtmpai18QC/file435409f1c2]
  gender_ind sex
       <int> <chr>
1          1 M
2          2 F
```

```
> head(select(sample.tracking.db, .tables="clinical"))
```

```
# Source:   lazy query [?? x 7]
# Database: sqlite 3.22.0 [/tmp/Rtmpai18QC/file435409f1c2]
  clinical_ind sample_id   age status var_wave_1 var_wave_2 gender_ind
         <int>     <int> <int>  <int>      <dbl>      <dbl>      <int>
1            1         1    19      1    -0.0520     -0.645          1
2            2         2    18      0    -1.75        0.165          2
3            3         3    17      0     0.0993      0.439          2
4            4         4     6      1    -0.572       0.883          1
5            5         5    10      0    -0.974      -2.05           2
6            6         6     8      1    -0.180      -1.64           1
```

```
> select(sample.tracking.db, .tables=c("clinical", "gender"))
```

```
# Source:   lazy query [?? x 8]
# Database: sqlite 3.22.0 [/tmp/Rtmpai18QC/file435409f1c2]
   clinical_ind sample_id   age status var_wave_1 var_wave_2 gender_ind sex
          <int>     <int> <int>  <int>      <dbl>      <dbl>      <int> <chr>
 1            1         1    19      1    -0.0520     -0.645          1 M
 2            2         2    18      0    -1.75        0.165          2 F
 3            3         3    17      0     0.0993      0.439          2 F
 4            4         4     6      1    -0.572       0.883          1 M
 5            5         5    10      0    -0.974      -2.05           2 F
 6            6         6     8      1    -0.180      -1.64           1 M
 7            7         8    13      1    -1.99        1.05           1 M
 8            8        10     6      0     0.117       0.715          1 M
 9            9        11    20      1    -0.893       0.917          1 M
```

```
10          10        12    20      1     0.334      -2.66           1 M
# ... with more rows

>
```

It is important to note that currently specifying a relationship between two tables in this manner will result in the 'to' table being limited to only those values in common between the specified columns in the two tables. So do not use this approach unless you expect that the two tables are consistent with each other.

3 Variant Call Format Database

One of the the most useful features of `poplite` is that databases can be created from any R object that can be coerced to a `data.frame` and they can be populated iteratively. This more complex example involves the parsing and loading of a variant call format (VCF) file. This type of file was devised to encode DNA variations from a reference genome for a given set of samples. In addition, it can also provide numerical summaries on the confidence that the variant exists as well as other summary stats and annotations. For more details, see the following specifications: `http://ga4gh.org/#/fileformats-team`. As parsing is already provided by the `VariantAnnotation` package [Obenchain et al., 2014] in Bioconductor [Gentleman et al., 2004], we will focus on the population of a database using this data. We will use the example dataset from the `VariantAnnotation` package and read it into memory as a `VCF` object as shown in the `VariantAnnotation` vignette.

The first table we will create is a representation of the chromosome, location and reference positions where at least one variant was observed. To do this we will create a function that takes a `VCF` object as input and returns a `data.frame`. This function and a subset of the `VCF` object is passed to the `makeSchemaFromFunction` helper function. What this helper function does is execute the desired function on the provided `VCF` object, determine the table structure from the returned `data.frame` and package the results in a `TableSchemaList` object. It is important that the name(s) for the input objects match between the provided function, `makeSchemaFromFunction` and ultimately `populate`.

```
> populate.ref.table <- function(vcf.obj)
+ {
+     ref.dta <- cbind(
+                      seqnames=as.character(seqnames(vcf.obj)),
+                      as.data.frame(ranges(vcf.obj))[,c("start", "end")],
+                      ref=as.character(ref(vcf.obj)),
+                      stringsAsFactors=FALSE
+                      )
+     return(ref.dta)
+ }
> vcf.sc <- makeSchemaFromFunction(populate.ref.table, "reference", vcf.obj=vcf[1:5])
>
```

Each variant position in our 'reference' table can have multiple alternatives (termed alleles) which were determined by the genotyping program which generated the VCF file. The 'alleles' table can be formed as follows:

```
> populate.allele.table <- function(vcf.obj)
+ {
+     exp.obj <- expand(vcf.obj)
+     ref.dta <- cbind(
+                      seqnames=as.character(seqnames(exp.obj)),
+                      as.data.frame(ranges(exp.obj))[,c("start", "end")],
+                      ref=as.character(ref(exp.obj)),
+                      alt=as.character(alt(exp.obj)),
+                       stringsAsFactors=FALSE
+                       )
+     return(ref.dta)
+ }
> allele.sc <- makeSchemaFromFunction(populate.allele.table, "alleles", vcf.obj=vcf[1:5])
> vcf.sc <- poplite::append(vcf.sc, allele.sc)
>
```

Finally we can form a table that records the number of alternative alleles for each sample. Implementation Note: the 'allele_count' column is very simple due to fact that this example dataset only contains one alternative allele for each variant. Multi-alleleic cases would have to be dealt with differently as we would need to be sure that the appropriate allele was counted for each genotype.

```
> populate.samp.alt.table <- function(vcf.obj)
+ {
+     temp.vrange <- as(vcf.obj, "VRanges")
+
+     ret.dta <- cbind(
+                         seqnames=as.character(seqnames(temp.vrange)),
+                         as.data.frame(ranges(temp.vrange))[,c("start", "end")],
+                         ref=ref(temp.vrange),
+                         alt=alt(temp.vrange),
+                         sample=as.character(sampleNames(temp.vrange)),
+                         allele_count=sapply(strsplit(temp.vrange$GT, "\\|"),
+                                 function(x) sum(as.integer(x), na.rm=T)),
+                         stringsAsFactors=F
+                         )
+
+     return(ret.dta[ret.dta$allele_count > 0,])
+ }
> geno.all.sc <- makeSchemaFromFunction(populate.samp.alt.table, "sample_alleles", vcf.obj=vcf[1:5])
> vcf.sc <- poplite::append(vcf.sc, geno.all.sc)
>
```

Our relationships can be specified as before. One wrinkle here is that there is a dependence in the table structures that is more complicated than in the 'Sample Tracking Database' section above. The first relationship statement below will create a final 'alleles' table with the primary key of 'reference' and the 'alt' column. As 'sample_allleles' contains the full column structure of 'alleles' plus the 'sample' and 'allele_count' columns it has to be first joined with reference to retrieve its primary key, then joined with the new 'alleles' table to get 'alleles' primary key with the final table resembling: 'alt_ind', 'sample' and 'allele_count'. Just as '.' on the left hand side indicates the primary key of the 'from' table the '.reference' variable indicates the primary key of the reference table.

```
> relationship(vcf.sc, from="reference", to="alleles") <- .~seqnames+start+end+ref
> relationship(vcf.sc, from="reference", to="sample_alleles") <- .~seqnames+start+end+ref
> relationship(vcf.sc, from="alleles", to="sample_alleles") <- .~.reference+alt
>
```

Now that our schema is complete, we can populate our three tables in the database. In this case, we can populate the entire database with one statement, however for demonstration purposes we will divide the `VCF` object into several pieces and populate them one at a time to simulate reading from a large file in chunks or iterating over several files. Also note that the 'constraint<-' method is available to provide a mechanism to enforce uniqueness of a subset of columns for a given table which is especially useful if the database is being populated from many files which may have duplicate data.

```
> vcf.db <- Database(vcf.sc, tempfile())
> populate(vcf.db, vcf.obj=vcf[1:1000])
> populate(vcf.db, vcf.obj=vcf[1001:2000])
> pop.res <- as.data.frame(poplite::select(vcf.db, .tables=tables(vcf.db)))
>
> sessionInfo()

R version 3.6.0 (2019-04-26)
Platform: x86_64-pc-linux-gnu (64-bit)
Running under: Debian GNU/Linux 9 (stretch)

Matrix products: default
BLAS:   /usr/local/lib/R/lib/libRblas.so
LAPACK: /usr/local/lib/R/lib/libRlapack.so

locale:
[1] C

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

other attached packages:
[1] RSQLite_2.1.1   poplite_0.99.23 DBI_1.0.0       dplyr_0.8.0.1  

loaded via a namespace (and not attached):
 [1] igraph_1.2.4.1   Rcpp_1.0.1       magrittr_1.5     tidyselect_0.2.5
 [5] bit_1.1-14       R6_2.4.0         rlang_0.3.4      fansi_0.4.0     
 [9] blob_1.1.1       tools_3.6.0      utf8_1.1.4       cli_1.1.0       
[13] dbplyr_1.4.0     bit64_0.9-7      lazyeval_0.2.2   assertthat_0.2.1
[17] digest_0.6.18    tibble_2.1.1     crayon_1.3.4     purrr_0.3.2     
[21] memoise_1.1.0    glue_1.3.1       compiler_3.6.0   pillar_1.3.1    
[25] pkgconfig_2.0.2 
```

References

Robert C Gentleman, Vincent J Carey, Douglas M Bates, Ben Bolstad, Marcel Dettling, Sandrine Dudoit, Byron Ellis, Laurent Gautier, Yongchao Ge, Jeff Gentry, et al. Bioconductor: open software development for computational biology and bioinformatics. *Genome biology*, 5(10):R80, 2004.

Valerie Obenchain, Michael Lawrence, Vincent Carey, Stephanie Gogarten, Paul Shannon, and Martin Morgan. Variantannotation: a bioconductor package for exploration and annotation of genetic variants. *Bioinformatics*, 30(14):2076–2078, 2014. doi: 10.1093/bioinformatics/btu168. URL `http://bioinformatics.oxfordjournals.org/content/30/14/2076.abstract`.

Hadley Wickham and Romain Francois. *dplyr: A Grammar of Data Manipulation*, 2014. URL `http://CRAN.R-project.org/package=dplyr`. R package version 0.3.0.2.