

The Genominator User Guide

James Bullard, Kasper Daniel Hansen

December 31, 2009

1 Introduction

Overview

The *Genominator* package provides an interface to storing and retrieving genomic data, together with some additional functionality aimed at high-throughput sequence data. The intent is that retrieval and summarization will be fast enough to enable online experimentation with the data.

We have used the package to analyze tiling arrays and (perhaps more appropriate) RNA-Seq data consisting of more than 400 million reads.

The canonical use case at the core of the package is summarizing the data over a large number of genomic regions. The standard example is for each annotated exon in human, count the number of reads that lands in that exon, for all experimental samples.

Data is stored in a SQLite database, and as such the package makes it possible to work with very large datasets in limited memory. However, working with SQLite databases is limited by I/O (disk speed), and substantial performance gains are possible by using a fast disk.

Limitations

While data may be stored in different ways using the experimental data model described below, typically (for sequencing data) we associate each read to a single location in the genome. This means that the package does not currently support paired-end reads, nor dealing with reads that span exon-exon junctions. Work is being done to include these important cases.

As mentioned above, the package uses a SQLite backend and the speed of its functionality is primarily limited by disk speed, which is unlike many other R packages. There is no gain in having multiple cores available and there is substantial slow down when accessing the data over a networked drive.

2 Data model

The core functionality of the package is the analysis of *experimental* data using genomic *annotation*, such as counting the number of reads falling in annotated exons. In the following we describe what we mean by experimental and annotation data.

2.1 Experimental data

The package utilizes *RSQLite* to store the data corresponding to an experiment. The SQL data model is:

```
chr INTEGER, strand INTEGER (-1,0,1), location INTEGER, [name NUMERIC]*
```

Specifically it means that each data unit has an associated (non-empty) genomic location triplet, namely **chr** (chromosome coded as an integer), **strand** (one of -1, 0, or 1 representing reverse strand, no strand information, and forward strand respectively) as well as a **location**. We also allow an unlimited number of additional numeric variables. (The requirement that the additional variables be numeric is purely an optimization.) There is no requirement that the location triplet (chr, strand, location) only occurs once in the data.

Examples are

```
chr INTEGER, strand INTEGER (-1,0,1), location INTEGER, chip_1 REAL, chip_2 REAL
chr INTEGER, strand INTEGER (-1,0,1), location INTEGER, counts INTEGER
```

The first data model could describe data from two tiling arrays, and in that case there would typically only be a single occurrence of each location triplet (chr, strand, location).

The second data model could describe

- data from a single sequencing experiment, each row corresponding to a read, such as

```
chr, strand, location, counts
1, 1, 1, 1
1, 1, 1, 1
1, 1, 2, 1
```

(here two reads map to the same location)

- data from a single sequencing experiment, but where the reads have been *aggregated* such that each row corresponds to the number of reads mapped to that location.

```
chr, strand, location, counts
1, 1, 1, 2
1, 1, 1, 1
```

2.2 Annotation data

Unlike experimental data, which is stored in an SQLite database, we represent annotation as an R `data.frame` with the following columns:

```
chr integer, strand integer (-1L,0L,1L), start integer, end integer, [name class]*
```

As in the experimental data representation, strand information is encoded as -1 for Reverse/Minus Strand, 0 for No Strand Information Available/Relevant, and 1 for Forward/Plus Strand. Each row is typically called a region (or region of interest, abbreviated ROI).

Since each region is consecutive, a transcript with multiple exons needs to be represented by multiple regions and one would typically have a column indicating transcript id, linking the different regions together. This data model is very similar to the genomic feature format (GFF).

A common example is

```
chr integer, strand integer (-1L,0L,1L), start integer, end integer, feature factor
```

3 Overview

There are 3 broad classes of functions within *Genominator*: functions that import and transform data, functions that retrieve and summarize data and finally functions that operate on retrieved data (focused on analysis of next generation sequencing data).

In the next two sections we will generate experimental data and annotation data for use in later examples.

3.1 Creating Experimental Data

We are going to walk through a very simple example using simulated experimental data to present the data import pipeline. This example uses a verbose setting of `TRUE` to illustrate activities performed in the SQLite databases. The example data will be used later in the vignette to illustrate other aspects of the package.

For an example of importing “real” next generation sequence data, see the companion vignette on working with the *ShortRead* package.

The data can be thought of as next generation sequencing data (N number of reads), in an organism with 16 chromosomes and a small number of annotated regions (K regions).

```
> library(Genominator)
> options(verbose = TRUE)
> set.seed(123)
> N <- 100000L
> K <- 100L
> df <- data.frame(chr = sample(1:16, size = N, replace = TRUE),
+   location = sample(1:1000, size = N, replace = TRUE),
+   strand = sample(c(1L, -1L), size = N, replace = TRUE))
> head(df)
```

	chr	location	strand
1	5	603	-1
2	13	23	-1
3	7	821	-1
4	15	37	-1
5	16	236	1
6	1	871	-1

```
> eDataRaw <- importToExpData(df, filename = "my.db", tablename = "ex_tbl",
+   overwrite = TRUE)
```

Writing table: 0.25 sec

Creating index: 0.29 sec

```
> eDataRaw
```

table: ex_tbl

database file: my.db

index columns: chr location strand

```
mode: w
schema:
  chr location strand
"INTEGER" "INTEGER" "INTEGER"
```

```
> head(eDataRow)
```

```
  chr location strand
1    1         1    -1
2    1         1    -1
3    1         1    -1
4    1         1    -1
5    1         1    -1
6    1         1    -1
7    1         1    -1
8    1         1     1
9    1         1     1
10   1         2    -1
```

The `df` object contains unsorted reads. `eDataRow` is an example of an `ExpData` object, the core object in Genominator. Such an object essentially points to a specific table in a specific database (in SQLite a database is a file). The data in `eDataRow` is ordered along the genome (unlike `df`), but there may be multiple rows with the same genomic location. The argument `overwrite = TRUE` indicates that if the table already exists in the database, overwrite it. This can be handy for scripts and vignettes.

The `eDataRow` has a number of slots related to an internal bookkeeping of database connection. The `index columns` indicates what columns of the database are indexed. For “normal” uses this will always be (chr, location, strand). The `mode` indicates whether the database is in read or write mode (write implies read).

In a normal pipeline, the first thing we do is aggregate the reads. With default settings, this means counting the number of reads starting at each (chr, location, strand). The resulting database has only one row with a given (chr, location, strand).

```
> eData <- aggregateExpData(eDataRow, tablename = "counts_tbl",
+   deleteOriginal = FALSE, overwrite = TRUE)
```

```
Creating table: counts_tbl: 0 sec
inserting: 0.09 sec
creating index: 0.08 sec
```

```
> eData
```

```
table: counts_tbl
database file: my.db
index columns: chr location strand
mode: w
schema:
  chr location strand counts
"INTEGER" "INTEGER" "INTEGER" "INTEGER"
```

```
> head(eData)
```

	chr	location	strand	counts
1	1	1	-1	7
2	1	1	1	2
3	1	2	-1	1
4	1	2	1	2
5	1	3	-1	3
6	1	3	1	4
7	1	4	-1	4
8	1	4	1	3
9	1	5	-1	1
10	1	5	1	5

The return object is (as always) an `ExpData` object pointing to the table that was *created*. Note the addition of the `counts` column.

The input `ExpData` object points to table `ex_tbl` in database `my.db`. The output `ExpData` object (currently) always uses the same database as the input object, possibly with a different name (in this case `counts_tbl`). All functions that manipulate databases have the arguments `overwrite` and `deleteOriginal`. If `deleteOriginal` is `TRUE`, the original table (in this case `ex_tbl`) is deleted. If `tablename` is `NULL` (default), the function does a destructive in-place modification of the table.

It is possible to break `ExpData` objects. For example, if we had used the `aggregateExpData` function with `deleteOriginal = TRUE`, the table that `eDataRow` points to would have been deleted. Or, if the function had been used with `tablename = NULL` (default), both `eDataRow` and `eData` would point to the same table in the database, but the schema recorded in `eDataRow` during instantiation would be out of date because it would not include the `counts` column. While this may seem problematic, it has not been cause for much concern. Remember, that `ExpData` objects are very cheap to create, so if something seems to break, delete and recreate it. With a bit of familiarity, this problem can be avoided. In general, we highly recommend carrying out the creation/manipulation of data in a script separate from the analysis, since creation/manipulation requires write access, whereas analysis is read only.

Each `ExpData` object has a mode that indicates whether the database is in read or write, which also implies read, mode. The `eDataRow` and `eData` objects created above had a 'write' mode. To prevent unwanted modifications to the database, we will instantiate a new `eData` object in 'read' only mode.

```
> eData <- ExpData("my.db", tablename = "counts_tbl")
```

```
> eData
```

```
table: counts_tbl
database file: my.db
index columns: chr location strand
mode: r
schema:
      chr location strand counts
"INTEGER" "INTEGER" "INTEGER" "INTEGER"
```

```
> head(eData)
```

	chr	location	strand	counts
1	1	1	-1	7
2	1	1	1	2
3	1	2	-1	1
4	1	2	1	2
5	1	3	-1	3
6	1	3	1	4
7	1	4	-1	4
8	1	4	1	3
9	1	5	-1	1
10	1	5	1	5

This used the constructor function `ExpData`, which is a standard way to instantiate `ExpData` objects in a new session.

It is possible to use the normal `[]` and `$` operators on `ExpData` objects (although they do not have rownames). This is rarely necessary, and the return objects may be massive.

```
> head(eData$chr)
```

	chr
1	1
2	1
3	1
4	1
5	1
6	1

```
> eData[1:3, 1:2]
```

	chr	location
1	1	1
2	1	1
3	1	2

3.2 Creating Annotation

We now create a suitable annotation object. As described in earlier sections, annotation consists of consecutive genomic regions that may or may not be overlapping.

```
> annoData <- data.frame(chr = sample(1:16, size = K, replace = TRUE),
+   strand = sample(c(1L, -1L), size = K, replace = TRUE),
+   start = (st <- sample(1:1000, size = K, replace = TRUE)),
+   end = st + rpois(K, 75), feature = c("gene", "intergenic")[sample(1:2,
+   size = K, replace = TRUE)])
> rownames(annoData) <- paste("elt", 1:K, sep = ".")
> head(annoData)
```

	chr	strand	start	end	feature
elt.1	13	-1	460	521	gene
elt.2	4	-1	332	405	gene
elt.3	13	1	177	236	gene
elt.4	3	1	390	472	gene
elt.5	15	1	130	203	gene
elt.6	1	1	885	964	intergenic

In this example, the `annoData` object needs to have distinct row names in order to maintain the link between annotation and returned data structures from the *Genominator* API.

Also the `strand` column needs to have values in $\{-1, 0, 1\}$, and the `chr` column needs to have integer values. When you access “real” annotation, you will often need a post-processing step where the annotation gets massaged into this format. See the additional vignette on working with the *ShortRead* package for a real-life example.

4 Creating and managing data

For illustrative purposes, we generate another set of data:

```
> df2 <- data.frame(chr = sample(1:16, size = N, replace = TRUE),
+   location = sample(1:1000, size = N, replace = TRUE),
+   strand = sample(c(1L, -1L), size = N, replace = TRUE))
> eData2 <- aggregateExpData(importToExpData(df2, filename = "my.db",
+   tablename = "ex2", overwrite = TRUE))
```

```
Writing table: 0.24 sec
Creating index: 0.29 sec
Creating table: __tmp_9508: 0 sec
inserting: 0.1 sec
dropping original table: 0.02 sec
renaming table: 0.02 sec
creating index: 0.09 sec
```

as well as re-doing the aggregation performed previously (with a new tablename)

```
> eData1 <- aggregateExpData(importToExpData(df, filename = "my.db",
+   tablename = "ex1", overwrite = TRUE))
```

```
Writing table: 0.25 sec
Creating index: 0.3 sec
Creating table: __tmp_7215: 0.02 sec
inserting: 0.09 sec
dropping original table: 0.02 sec
renaming table: 0 sec
creating index: 0.1 sec
```

4.1 Aggregation

Aggregation refers to aggregation over rows of the database. This is typically used for sequencing data and is typically employed in order to go from a “one row, one read” type representation to a “one row, one genomic location with an associated number of reads”. The default arguments creates a new column `counts` using an “aggregator” that is the number of times a genomic location occurs in the data. Aggregation has also been discussed in an earlier section.

4.2 Merging

It is natural to store a number of experimental replicates in columns of a table. However, it is often the case that we receive the data in chunks over time, and that merging new values with old values is not trivial. For this reason we provide a `joinExpData` function to bind two tables together.

Essentially, merging consists of placing two (or more) columns next to each other. It is (somewhat) clear that (in general) it makes the most sense to merge two datasets when each dataset has only a single occurrence of each genomic location. Otherwise, how would we deal with/interpret the case where a genomic location occurs multiple times in each dataset? For that reason, joining two datasets typically happens after they have been aggregated.

It is possible to merge/join the datasets in R and then subsequently use the import facility to store the resulting object in a database. In general, that approach is less desirable because 1) it is slow(er) and 2) it requires all datasets to be present in memory.

```
> eDataJoin <- joinExpData(list(eData1, eData2), fields = list(ex1 = c(counts = "counts_1"
+      ex2 = c(counts = "counts_2")), tablename = "allcounts")
```

```
Creating union: 0.05 sec
```

```
Left outer join with table ex1: 0.14 sec
```

```
Left outer join with table ex2: 0.14 sec
```

```
Indexing: 0.11 sec
```

```
> head(eDataJoin)
```

	chr	location	strand	counts_1	counts_2
1	1	1	-1	7	2
2	1	1	1	2	3
3	1	2	-1	1	2
4	1	2	1	2	5
5	1	3	-1	3	3
6	1	3	1	4	3
7	1	4	-1	4	NA
8	1	4	1	3	1
9	1	5	-1	1	3
10	1	5	1	5	1

In this example both `eData1` and `eData2` have a column named `counts` that we rename in the resulting object using the `fields` argument. Also missing values are introduced when the locations in one object are not present in the other. Finally, the `joinExpData` function supports joining an arbitrary number of `ExpData` objects.

We can examine the result using `summarizeExpData` (described later)


```
> summarizeExpData(eDataJoin, fxs = c("total", "avg", "count"))
```

```
fetching summary: 0.05 sec
```

total(counts_1)	total(counts_2)	avg(counts_1)	avg(counts_2)
1.000000e+05	1.000000e+05	3.267760e+00	3.267547e+00
count(counts_1)	count(counts_2)		
3.060200e+04	3.060400e+04		

Because of the way we store the data, the “total” column will be the total number of read, the “count” column will be the number of bases where a read starts, and the “avg” column will be average number of reads at a given genomic location (removing the genomic locations that were not sequenced).

4.3 Collapsing

Another common operation is collapsing data across columns. One use would be to join to experiments where the same sample was sequenced. One advantage of collapsing the data in this case is speed.

Collapsing is most often done using summation (the default):

```
> head(collapseExpData(eDataJoin, tablename = "collapsed",
+   collapse = "sum", overwrite = TRUE))
```

```
CREATE TABLE collapsed (chr INTEGER, location INTEGER, strand INTEGER, COL)
```

```
creating table: 0.11 sec
```

```
INSERT INTO collapsed SELECT chr, location, strand, CAST(TOTAL(counts_1)AS INTEGER)+CAST(TOTAL(counts_2)AS INTEGER)
```

```
inserting data: 0.11 sec
```

```
creating index: 0.09 sec
```

	chr	location	strand	COL
1	1	1	-1	9
2	1	1	1	5
3	1	2	-1	3
4	1	2	1	7
5	1	3	-1	6
6	1	3	1	7
7	1	4	-1	4
8	1	4	1	4
9	1	5	-1	4
10	1	5	1	6

But it could also be done using an “average” or a “weighted average” (weighted according to sequencing effort).

```
> head(collapseExpData(eDataJoin, tablename = "collapsed",
+   collapse = "weighted.avg", overwrite = TRUE))
```

```
fetching summary: 0.01 sec
```

```
CREATE TABLE collapsed (chr INTEGER, location INTEGER, strand INTEGER, COL)
```

```
creating table: 0 sec
```

```
INSERT INTO collapsed SELECT chr, location, strand, TOTAL(counts_1) * 0.5+TOTAL(counts_2) * 0.5
```

```
inserting data: 0.11 sec
```

```
creating index: 0.08 sec
```

```
chr location strand COL
1      1      1     -1 4.5
2      1      1      1 2.5
3      1      2     -1 1.5
4      1      2      1 3.5
5      1      3     -1 3.0
6      1      3      1 3.5
7      1      4     -1 2.0
8      1      4      1 2.0
9      1      5     -1 2.0
10     1      5      1 3.0
```

```
> head(collapseExpData(eDataJoin, tablename = "collapsed",
+ collapse = "avg", overwrite = TRUE))
```

```
CREATE TABLE collapsed (chr INTEGER, location INTEGER, strand INTEGER, COL)
```

```
creating table: 0 sec
```

```
INSERT INTO collapsed SELECT chr, location, strand, (TOTAL(counts_1)+TOTAL(counts_2))/2 FROM
```

```
inserting data: 0.11 sec
```

```
creating index: 0.07 sec
```

```
chr location strand COL
1      1      1     -1 4.5
2      1      1      1 2.5
3      1      2     -1 1.5
4      1      2      1 3.5
5      1      3     -1 3.0
6      1      3      1 3.5
7      1      4     -1 2.0
8      1      4      1 2.0
9      1      5     -1 2.0
10     1      5      1 3.0
```

In this case setting `collapse` equal to `avg` or `weighted.avg` respectively yields the exact same result, since the two experiments have the same number of reads.

5 Interface

In this section we describe the core functionality of the *Genominator* package.

We will use two examples: one `ExpData` consisting of a single (aggregated) experiment and one `ExpData` consisting of two aggregated, joined experiments.

```
> eData <- ExpData("my.db", tablename = "counts_tbl", mode = "r")
> eDataJoin <- ExpData("my.db", tablename = "allcounts",
+ mode = "r")
```

5.1 Summarizing experimental data

We can use the function `summarizeExpData` to summarize `ExpData` objects. This function does not utilize annotation, so the summarization is in some sense “global”. The call to generate the total number of counts, i.e. the number of reads, in column “counts” is

```
> ss <- summarizeExpData(eData, what = "counts")
```

```
fetching summary: 0.02 sec
```

```
> ss
```

```
[1] 1e+05
```

The `what` argument is present in many of the following functions. It refers to which columns are being used, and in general the default depends on the type of function. If the function summarizes data, the default is “all columns, except the genomic location columns”, whereas if the function retrieves data, the default is “all columns”.

We can customize the summary by specifying the name of any SQLite function (www.sqlite.org/lang_aggf) in the `fxs` argument

```
> summarizeExpData(eData, what = "counts", fxs = c("MIN",
+          "MAX"))
```

```
fetching summary: 0.02 sec
```

```
MIN(counts) MAX(counts)
```

```
1          14
```

This yields the maximum/minimum number of reads mapped to a single location. The minimum number of reads is not zero because we only store locations associated with reads.

5.2 Selecting a region

We can access genomic data in a single genomic region using the function `getRegion`.

```
> reg <- getRegion(eData, chr = 2L, strand = -1L, start = 100L,
+          end = 105L)
```

```
SELECT * FROM counts_tbl WHERE chr = 2 AND (strand IN (-1) OR strand = 0) AND location between
fetching region query: 0.01 sec
```

```
> class(reg)
```

```
[1] "data.frame"
```

```
> reg
```

	chr	location	strand	counts
1	2	100	-1	3
2	2	101	-1	5
3	2	102	-1	4
4	2	103	-1	6
5	2	104	-1	3
6	2	105	-1	4

It is possible exclude either `start` or `end` in which case the default values are 0 and 1e12.

5.3 Using annotation with experimental data

The two previous sections show useful functions, but in general we want to summarize data over many genomic regions simultaneously, in order to

- Summarize regions (means, lengths, sums, etc)
- Fit models on each region
- Perform operations over classes of regions (genes, intergenic regions, ncRNAs)

There are essentially two different strategies for this: retrieve the data as a big object and then use R functions to operate on the data (e.g. `splitByAnnotation`) or perform some operation on the different regions in the database (e.g. `summarizeByAnnotation`). The first approach is more flexible, but also slower and requires more memory. The second approach is faster, but limited to operations that can be expressed in SQL.

First, we demonstrate how to summarize over regions of interest. Here we are going to compute the SUM and COUNT of each region, which tell us the total number of sequencing reads at each location and the number of unique locations that were read respectively.

```
> head(summarizeByAnnotation(eData, annoData, what = "counts",
+   fxs = c("COUNT", "TOTAL"), bindAnno = TRUE))
```

```
writing regions table: 0.01 sec
```

```
SELECT __regions__.id, COUNT(counts), TOTAL(counts) FROM __regions__ LEFT OUTER JOIN counts_
```

```
fetching summary table: 0.04 sec
```

	chr	strand	start	end	feature	COUNT(counts)	TOTAL(counts)
elt.1	13	-1	460	521	gene	58	193
elt.2	4	-1	332	405	gene	69	219
elt.3	13	1	177	236	gene	57	190
elt.4	3	1	390	472	gene	82	258
elt.5	15	1	130	203	gene	70	235
elt.6	1	1	885	964	intergenic	78	241

The result of `summarizeByAnnotation` is a `data.frame`. We use `bindAnno = TRUE` in order to keep the annotation as part of the result, which is often very useful.

The `fxs` argument takes the names of SQLite functions, see www.sqlite.org/lang_aggfunc.html. One important note regarding summation: the standard SQL function `SUM` handles the sum of only missing values, which in R is expressed as `sum(NA)`, differently from the SQLite specific function `TOTAL`. In particular, the `SUM` function returns a missing value and the `TOTAL` function returns a zero. This is relevant when summarizing over a genomic region containing no reads in one experiment, but reads in another experiment.

This next example computes, the number of reads mapping to the region for each annotated region, ignoring strand. Ignoring strand might be the right approach if the protocol does not retain strand information (e.g. the current standard protocol for Illumina RNA-Seq).

```
> head(summarizeByAnnotation(eDataJoin, annoData, , fxs = c("SUM"),
+   bindAnno = TRUE, preserveColnames = TRUE, ignoreStrand = TRUE))
```

writing regions table: 0.04 sec

```
SELECT __regions__.id, SUM(counts_1), SUM(counts_2) FROM __regions__ LEFT OUTER JOIN allcount
```

fetching summary table: 0.06 sec

	chr	strand	start	end	feature	counts_1	counts_2
elt.1	13	-1	460	521	gene	417	390
elt.2	4	-1	332	405	gene	469	434
elt.3	13	1	177	236	gene	367	367
elt.4	3	1	390	472	gene	519	518
elt.5	15	1	130	203	gene	463	489
elt.6	1	1	885	964	intergenic	501	484

Essentially, this output is the input to a simple differential expression analysis.

Note that if two regions in the annotation overlap, data falling in the overlap will be part of the end result for each region.

We can produce summarizes by category using the `splitBy` argument.

```
> res <- summarizeByAnnotation(eData, annoData, what = "counts",
+   fxs = c("TOTAL", "COUNT"), splitBy = "feature")
```

writing regions table: 0.01 sec

```
SELECT __regions__.id, TOTAL(counts), COUNT(counts) FROM __regions__ LEFT OUTER JOIN counts_
```

fetching summary table: 0.03 sec

```
> class(res)
```

```
[1] "list"
```

```
> lapply(res, head)
```

\$gene

	TOTAL(counts)	COUNT(counts)
elt.1	193	58
elt.2	219	69
elt.3	190	57
elt.4	258	82
elt.5	235	70
elt.7	301	86

\$intergenic

	TOTAL(counts)	COUNT(counts)
elt.6	241	78
elt.8	283	86
elt.9	221	68
elt.11	206	62
elt.13	228	82
elt.16	235	72

Finally, we might want to join the relevant annotation to the summaries using the `bindAnno` argument.

```
> res <- summarizeByAnnotation(eData, annoData, what = "counts",
+   fxs = c("TOTAL", "COUNT"), splitBy = "feature", bindAnno = TRUE)

writing regions table: 0.03 sec
SELECT __regions__.id, TOTAL(counts), COUNT(counts) FROM __regions__ LEFT OUTER JOIN counts_
fetching summary table: 0.03 sec

> lapply(res, head)

$gene
      chr strand start end feature TOTAL(counts) COUNT(counts)
elt.1  13     -1   460 521   gene           193           58
elt.2   4     -1   332 405   gene           219           69
elt.3  13      1   177 236   gene           190           57
elt.4   3      1   390 472   gene           258           82
elt.5  15      1   130 203   gene           235           70
elt.7   2      1    24 109   gene           301           86

$intergenic
      chr strand start end   feature TOTAL(counts) COUNT(counts)
elt.6   1      1   885 964 intergenic           241           78
elt.8  16     -1   729 820 intergenic           283           86
elt.9   6      1   656 726 intergenic           221           68
elt.11 14     -1   550 612 intergenic           206           62
elt.13  1     -1   116 201 intergenic           228           82
elt.16  8     -1   419 492 intergenic           235           72
```

(Both of these example might require `ignoreStrand = TRUE` in a real world application.)

Unfortunately, the `summarizeByAnnotation` function is only able to utilize the very small set of SQLite functions, essentially limiting the function to computing very simple summaries.

We will now examine the `splitByAnnotation` function that splits the data according to annotation and returns the “raw” data. The return object of this function may be massive depending on the size of the data and the size of the annotation. In general we advise to do as much computation as possible using SQLite (essentially using `summarizeByAnnotation`), but sometimes it is necessary to access the raw data. Leaving aside the problems with the size of the return object, `splitByAnnotation` is reasonably fast.

We start by only splitting on the annotated regions of type “gene”.

```
> dim(annoData[annoData$feature %in% "gene", ])

[1] 51  5

> a <- splitByAnnotation(eData, annoData[annoData$feature %in%
+   "gene", ])

writing region table: 0.01 sec
SELECT counts_tbl.* , __regions__.id FROM counts_tbl INNER JOIN __regions__ ON __regions__.c
fetching splits table: 0.04 sec
SELECT count(__regions__.id), __regions__.id FROM counts_tbl INNER JOIN __regions__ ON __reg
count query: 0.01 sec
performing split: 0 sec
```

```

> class(a)

[1] "list"

> length(a)

[1] 51

> names(a)[1:10]

[1] "elt.1" "elt.2" "elt.3" "elt.4" "elt.5" "elt.7" "elt.10"
[8] "elt.12" "elt.14" "elt.15"

> head(a[[1]])

  chr location strand counts
1  13      460     -1      4
2  13      461     -1      5
3  13      462     -1      2
4  13      463     -1      3
5  13      464     -1      4
6  13      465     -1      2

```

There are several notes here. For starters, the return object is named according to the rownames of the annotation data. Also, only annotated regions with data are represented in the return object, so in general the return object does *not* have an element for each annotated region. We provide several convenience functions for operating on the results of `splitByAnnotation`, which are discussed later.

Now we wish to compute a trivial function over the counts, such as a quantile.

```

> sapply(a, function(x) {
+   quantile(x[, "counts"], 0.9)
+ })[1:10]

elt.1.90% elt.2.90% elt.3.90% elt.4.90% elt.5.90% elt.7.90%
      5      5      6      5      5      6
elt.10.90% elt.12.90% elt.14.90% elt.15.90%
      6      5      5      5

```

Often we wish to use the annotation information when operating on the data in each region. The `applyMapped` function makes this easy by appropriately matching up the annotation and the data. Essentially, this function ensures that you are applying the right bit of annotation to the correct chunk of data.

```

> applyMapped(a, annoData, FUN = function(region, anno) {
+   counts <- sum(region[, "counts"])
+   length <- anno$end - anno$start + 1
+   counts/length
+ })[1:10]

```

```

$elt.1
[1] 3.112903

$elt.2
[1] 2.959459

$elt.3
[1] 3.166667

$elt.4
[1] 3.108434

$elt.5
[1] 3.175676

$elt.7
[1] 3.5

$elt.10
[1] 3.405063

$elt.12
[1] 2.207317

$elt.14
[1] 3.068493

$elt.15
[1] 3.090909

```

This example computes the average number of reads per base, taking into account that bases without data exists. Note that `FUN` needs to be a function of two arguments.

What we see is that some of our regions are not present. This is a byproduct of the fact that some of our regions have no data within their bounds. One can successfully argue that the result of the example above ought to include such regions with a value of zero (see below for this).

When our data sets are large, it is often more convenient and significantly faster to return only some columns.

```

> sapply(splitByAnnotation(eData, annoData, what = "counts"),
+        median)[1:10]

```

```
writing region table: 0.03 sec
```

```
SELECT counts_tbl.counts , __regions__.id FROM counts_tbl INNER JOIN __regions__ ON __region
```

```
fetching splits table: 0.04 sec
```

```
SELECT count(__regions__.id), __regions__.id FROM counts_tbl INNER JOIN __regions__ ON __reg
```

```
count query: 0.02 sec
```

```
performing split: 0 sec
```


elt.1	elt.2	elt.3	elt.4	elt.5	elt.6	elt.7	elt.8	elt.9	elt.10
3	3	3	3	3	3	3	3	3	3

Often we wish to “fill” in missing regions. In the case of a coding sequence there may be bases that have no reads, and so these bases will not appear in our resulting object. We can “expand” a region to include these bases by filling in 0 reads for them. There are different ways to do this expansion. For convenience, data are stratified by strand. Therefore expansion will produce a list-of-lists, where each sublist has possibly two elements corresponding to each strand. If the original annotation query is stranded, then expansion will produce a list, where each sublist only has one element. Finally, we provide a feature to collapse across strand for the common use case of combining reads occurring on either strand within the region. In this case the return value is a list, where each element is an expanded matrix representing the reads that occurred on either strand.

```
> x1 <- splitByAnnotation(eData, annoData, expand = TRUE,
+   ignoreStrand = TRUE)
```

```
writing region table: 0.02 sec
```

```
SELECT counts_tbl.chr, counts_tbl.location, counts_tbl.strand, counts_tbl.* , __regions__.id
```

```
fetching splits table: 0.11 sec
```

```
SELECT count(__regions__.id), __regions__.id FROM counts_tbl INNER JOIN __regions__ ON __reg
```

```
count query: 0.03 sec
```

```
performing split: 0.01 sec
```

```
> names(x1[[1]])
```

```
[1] "-1" "1"
```

```
> x2 <- splitByAnnotation(eData, annoData, expand = TRUE)
```

```
writing region table: 0.02 sec
```

```
SELECT counts_tbl.chr, counts_tbl.location, counts_tbl.strand, counts_tbl.* , __regions__.id
```

```
fetching splits table: 0.06 sec
```

```
SELECT count(__regions__.id), __regions__.id FROM counts_tbl INNER JOIN __regions__ ON __reg
```

```
count query: 0.03 sec
```

```
performing split: 0 sec
```

```
> names(x2[[1]])
```

```
[1] "-1"
```

```
> x3 <- splitByAnnotation(eData, annoData, expand = TRUE,
+   addOverStrand = TRUE)
```

```
writing region table: 0.03 sec
```

```
SELECT counts_tbl.chr, counts_tbl.location, counts_tbl.strand, counts_tbl.* , __regions__.id
```

```
fetching splits table: 0.05 sec
```

```
SELECT count(__regions__.id), __regions__.id FROM counts_tbl INNER JOIN __regions__ ON __reg
```

```
count query: 0.03 sec
```

```
performing split: 0.02 sec
```

```
> head(x3[[1]])
```

	chr	location	counts
[1,]	13	460	4
[2,]	13	461	5
[3,]	13	462	2
[4,]	13	463	3
[5,]	13	464	4
[6,]	13	465	2

Leaving the `splitByAnnotation` function, sometimes we want to compute summaries of higher level entities, such as genes, pseudogenes, and intergenic regions. We can label each genomic location with its annotation using the `mergeWithAnnotation` convenience function.

```
> mergeWithAnnotation(eData, annoData)[1:3, ]
```

```
writing regions table: 0.03 sec
```

```
SELECT * FROM counts_tbl INNER JOIN __regions__ ON __regions__.chr = counts_tbl.chr AND coun
```

```
fetching merge table: 0.04 sec
```

	chr	location	strand	counts	id	chr	start	end	strand	feature
1	13	460	-1	4	1	13	460	521	-1	gene
2	13	461	-1	5	1	13	460	521	-1	gene
3	13	462	-1	2	1	13	460	521	-1	gene

This will result in duplicated genomic locations in case a genomic location is annotated multiple times.

There are a number of parameters that can make this more natural.

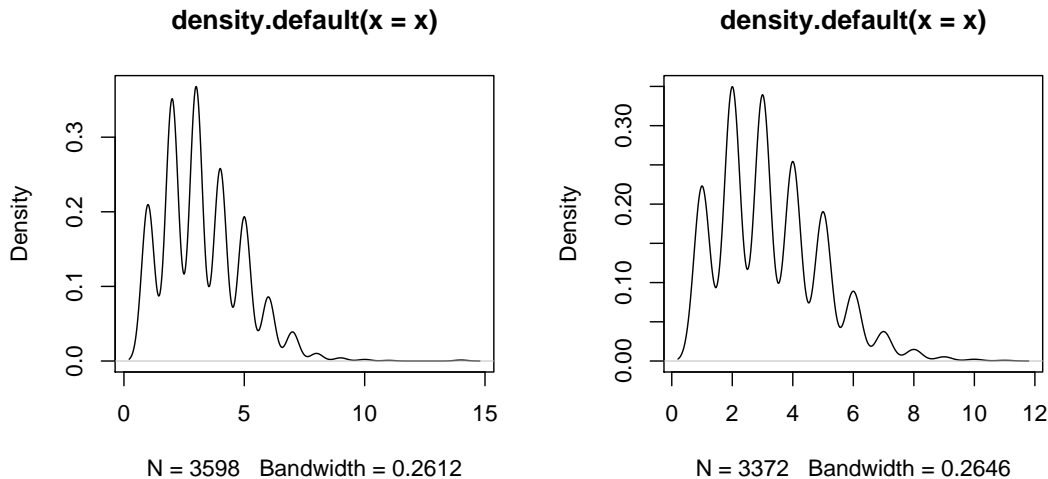
```
> par(mfrow = c(1, 2))
> x <- lapply(mergeWithAnnotation(eData, annoData, splitBy = "feature",
+   what = "counts"), function(x) {
+   plot(density(x))
+ })
```

```
writing regions table: 0.01 sec
```

```
SELECT counts,feature FROM counts_tbl INNER JOIN __regions__ ON __regions__.chr = counts_tbl
```

```
fetching merge table: 0.03 sec
```

```
splitting by: feature: 0 sec
```



6 Analysis tools

In the case of short read sequencing, the *Genominator* package offers a number of specific useful tools. They are presented in no particular order.

6.1 Coverage

The `computeCoverage` function can be used to assess the sequencing depth.

```
> coverage <- computeCoverage(eData, annoData, effort = seq(100,  
+   1000, by = 5), cutoff = function(e, anno, group) {  
+   e > 1  
+ })
```

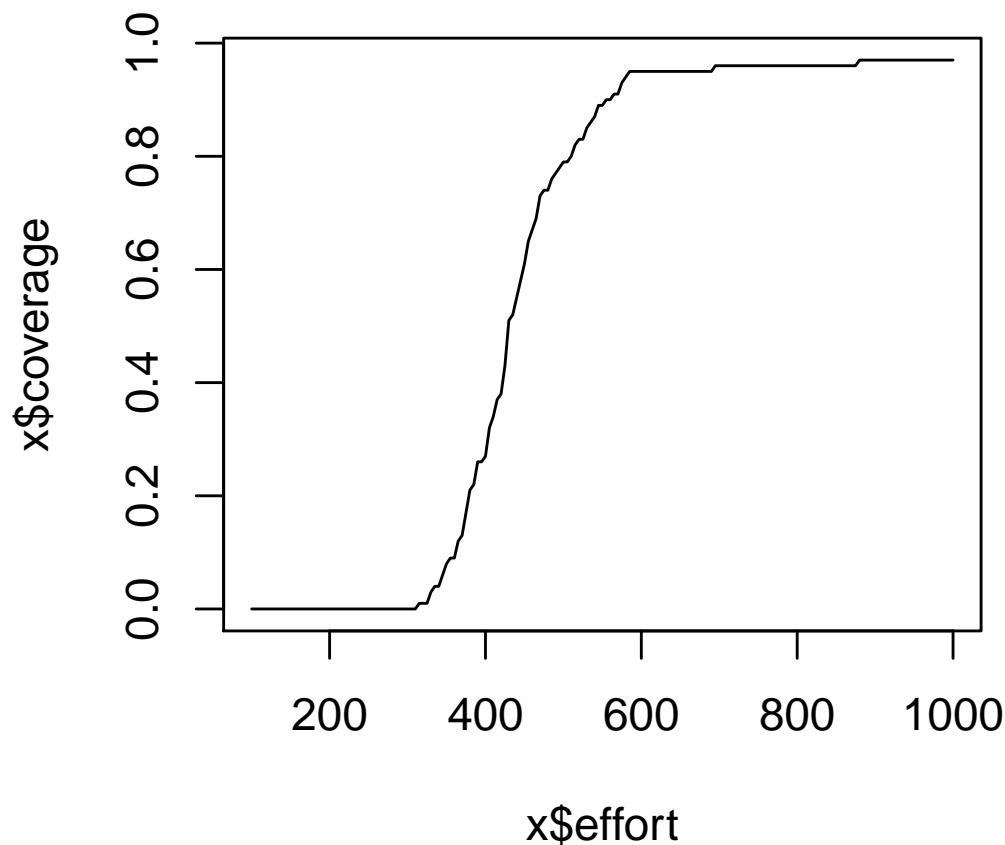
```
writing regions table: 0.02 sec
```

```
SELECT __regions__.id, TOTAL(counts) FROM __regions__ LEFT OUTER JOIN counts_tbl ON __region
```

```
fetching summary table: 0.05 sec
```

```
fetching summary: 0 sec
```

```
> plot(coverage, draw.legend = FALSE)
```



6.2 Statistical Functions: Goodness of Fit

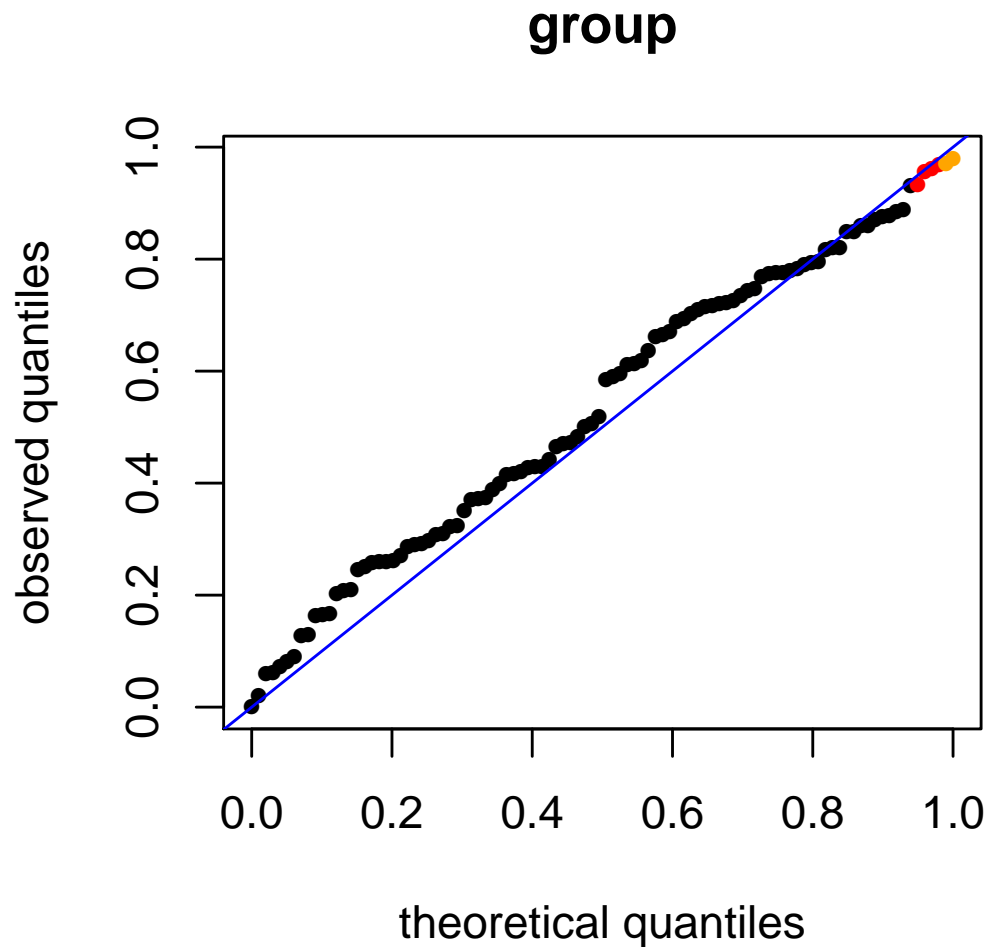
You can conduct a goodness of fit analysis to the Poisson model across lanes using the following function. The result is assessed by a QQ-plot against the theoretical distribution (of either the p-values or the statistic).

```
> plot(regionGoodnessOfFit(eDataJoin, annoData))
```

```
writing regions table: 0.01 sec
```

```
SELECT __regions__.id, TOTAL(counts_1), TOTAL(counts_2) FROM __regions__ LEFT OUTER JOIN all
```

```
fetching summary table: 0.05 sec
```



We can also do this for subsets of the data, for example within condition.

```
> plot(regionGoodnessOfFit(as.data.frame(matrix(rpois(1000,
+ 100), ncol = 10)), groups = rep(c("A", "B"), 5),
+ denominator = rep(1, 10)))
```

