# GROK User Guide
August 28, 2013

Kristian Ovaska

Contact: kristian.ovaska@helsinki.fi
Biomedicum Helsinki, University of Helsinki, Finland

# Contents

**1 Introduction** **1**

**2 Core concepts: region and region store** **1**
- 2.1 DNA region model . . . 1
- 2.2 Region interpretations: relation or interval . . . 3
- 2.3 Region store interface . . . 4
- 2.4 Region scores and aggregate functions . . . 5

**3 Region store implementations** **7**
- 3.1 File I/O . . . 7
- 3.2 In-memory indexed databases . . . 8
- 3.3 SQLite backend . . . 8
- 3.4 Filtering and transforming store . . . 9
- 3.5 Merging store . . . 9

**4 Installation and requirements** **11**
- 4.1 Unix . . . 11
  - 4.1.1 Source package . . . 11
  - 4.1.2 R bindings . . . 11
  - 4.1.3 Python bindings . . . 11
  - 4.1.4 Lua bindings . . . 12

**5 Using the R, Python and Lua APIs** **13**
- 5.1 Basic usage . . . 13
- 5.2 Data types . . . 13
- 5.3 Reading and writing regions in R, Python and Lua . . . 14
- 5.4 Iterating over regions . . . 14
- 5.5 Using region databases . . . 15
  - 5.5.1 Sorting regions in genomic order . . . 16
- 5.6 Overlap queries . . . 17
- 5.7 Set operations: union, intersection and difference . . . 18
- 5.8 Transforming regions . . . 19
- 5.9 Modifying region annotations . . . 20
- 5.10 Working with numeric region scores . . . 21
  - 5.10.1 Counting short reads along genome . . . 22
- 5.11 Method reference . . . 22
  - 5.11.1 Region store constructors . . . 23
  - 5.11.2 Region store accessors . . . 24
  - 5.11.3 Set operations . . . 25
  - 5.11.4 Region transformers . . . 26
  - 5.11.5 Region iterators and overlap queries . . . 27
  - 5.11.6 Region score manipulation . . . 28

**6 Using the command line interface** **29**

6.1 General usage . . . . . . . . . . . . . . . . . . . . . . . . . . . 29

6.2 File conversion . . . . . . . . . . . . . . . . . . . . . . . . . . . 29

6.3 Set operations . . . . . . . . . . . . . . . . . . . . . . . . . . . 30

**7 Using the C++ API** **31**

7.1 Accessing region attributes . . . . . . . . . . . . . . . . . . . . . . 32

7.2 Using region stores . . . . . . . . . . . . . . . . . . . . . . . . . 33

7.2.1 Iterating over regions and I/O . . . . . . . . . . . . . . . . . . 34

7.2.2 Using region databases . . . . . . . . . . . . . . . . . . . . . 35

7.3 Implementing new region stores . . . . . . . . . . . . . . . . . . . . 35

# 1 Introduction

GROK is a library for processing DNA regions, or intervals, obtained from high-throughput sequencing and other genomic applications. DNA regions represent sequencing shorts reads, gene locations, single nucleotide polymorphisms, ChIP-seq peaks and other chromosomal intervals. In addition to core attributes such as chromosomal coordinates, regions may contain arbitrary numeric or string annotations. Thus, DNA region processing covers many applications of high-throughput sequencing. Regions may be read from files such as BED, Wiggle, GFF, SAM and BAM, or they may be produced using custom algorithms written in C++ or other languages. Region operations supported by GROK include file format conversions, set operations (union, intersection and difference), overlap queries and filtering and transformation operations. GROK supports both in-memory and on-disk processing to enable operating on region sets that do not fit into memory.

GROK provides R, Python and Lua APIs for scripting and a C++ API for implementing custom algorithms on top of data structures and functions provided by GROK. GROK core is implemented in C++ with a focus on performance. The C++ layer has a modular structure and provides extension points for implementing support for new file formats and data structures.

# 2 Core concepts: region and region store

GROK architecture is based on two core abstractions: a DNA region model and a region store interface. Regions represent the sequencing data for processing, and region stores provide access to collections of regions. Complex operations are implemented using these abstractions.

## 2.1 DNA region model

A *region* is an annotated DNA interval that corresponds to one record in files such as BED, GFF and BAM. The region model is not specific to any single file format, but rather abstracts features found in commonly used genomic formats. Regions can be read from files but also created from custom analysis algorithms. Regardless of their origin, regions use the same data model and and be compared to each other and processed in similar manner. File format parser transforms regions into a standard format. However, regions from different origins (such as BED and BAM files) may support different sets of annotations.

Regions are represented as sets of attribute values, i.e., key-value pairs. Attributes are divided into core attributes, which are available for all regions regardless of their origin, and arbitrary number of optional annotations.

**Core attributes** These attributes encode the genomic coordinates and define region identity:

- `seqid`: Identifier that specifies the reference sequence for the region. Usually, this is a chromosome, such as `chr1`.
- `strand`: Strand of the genomic interval: forward (1), reverse (-1) or none (0). The none pseudo-strand is used when strand is not known or is not relevant.
- `left`: The leftmost position of the genomic interval, starting from zero.
- `length`: Length of the genomic interval. Length must not be negative, but zero is a valid length.
- `tag`: Numeric 64-bit value that has no semantic meaning but defines identity relationship between regions. Automatically generated.

The left and length attributes specify a half-open interval [`left`, `left+length`). Alternatively, we can define a pseudo-attribute `right` as `left+length` and the interval is then [`left`, `right`). The sequence ID defines the chromosome which this interval is part of, and strand defines whether the interval belongs to forward or reverse strand. The ”none” pseudo-strand is considered as a separate strand from forward and reverse strands.

The core attributes are a tuple (`seqid`, `strand`, `left`, `length`, `tag`) which defines region identity. Two regions with the same identity tuple are considered equal. This identity is used in set operations on regions.

**Annotations** In addition to core attributes, regions may have optional annotations, which are key-value pairs with numeric or string data as values. Examples of annotations include numeric scores (e.g., column 5 in BED format), string identifiers (e.g., column 4 in BED format) and nucleotide sequences for short reads. Each annotation is identified using its unique key. There may be any number of such annotations, as long as their names are unique. Regions from the same origin (e.g., same file format) share the same set of supported annotations. Annotation values do not affect region identity, since they are not part of the identity tuple. That is, two regions with the same core attributes but different annotations are still considered to be identical.

**Example** The following example shows three regions expressed with tuple notation. The tuples have values for sequence ID, strand, left coordinate, length, tag and annotations, in order.

```
reg1 = ("chr1", FORWARD, 0, 15, 0x1234, Score=5.3)
reg2 = ("chr1", FORWARD, 0, 15, 0x1234, Score=147, Label="region")
reg3 = ("chr2", NONE, 800, 32, 0x1234)
```

Regions reg1 and reg2 represent the interval [0, 15) in the forward strand of chromosome 1: these are the 15 first nucleotides of this chromosome. The regions are equal to each other because their identity tuples, including the hexadecimal tags, are equal. Region reg1 supports one numeric annotation, Score, and reg2 additionally supports a string annotation, Label. Regions reg1 and reg2 have different values for their annotations but this does not affect region identity: they are still equal to each other.

Region reg3 represents the non-stranded interval [800, 832) in chromosome 2. This region is not equal to reg1 or reg2 (even though its tag is incidentally the same) and does not define any annotations. Region reg3 would be non-equal to a region overlapping the same coordinates but having forward or reverse strand.

## 2.2 Region interpretations: relation or interval

DNA regions, represented by core attributes and annotations, can be semantically interpreted in two ways. The *relation interpretation* considers regions as relations (attribute tuples) in a database. Two regions are distinct if their identity tuples (see 2.1) are distinct, even if the regions have the same chromosomal coordinates (but their tags are different).

The alternative *interval interpretation* considers regions as intervals (sets of consecutive locations) located in a specific strand of a chromosome. Two regions sharing some chromosomal locations are said to be overlapping and hence have an implicit relationship between each other. GROK provides functionality to compute overlapping regions. In the interval interpretation, regions often have a score metric that associates a numeric value to the interval. The score can represent short read coverage, copy number, DNA methylation or other metric.

Both interpretations use the same attribute model but emphasize different aspects. Interval interpretation ignores the uniqueness tag used to distinguish regions with identical coordinates in the relation interpretation. Interval interpretation also places special emphasis on the score annotation.

## 2.3 Region store interface

GROK provides access to region collections through a *region store* interface. This interface defines methods for iterating over regions, writing new regions and, for some implementations, removing regions and querying regions that overlap specific genomic coordinates. There are multiple implementations of this interface, each supporting some subset of the methods. Region stores cover file I/O, intermediate storage for multi-pass algorithms and filtering and transformation operations. In combination with the DNA region model, the interface provides uniform access to these diverse use cases. Implementation details, such as the actual file formats or data structures used, are largely hidden from the user. Regions accessed using the same region store generally have the same set of annotations.

The methods defined by the region store interface are the following:

- `add`: Add a region to the database, or write (append) a region to a file.
- `iterator`: Provide access to each region in the collection one at a time. Order depends on the region store: regions may be sorted or unsorted.
- `get`: Fetch a specific region from the collection using region identity. Produce an empty result if the region is not present.
- `remove`: Remove a region from the database using region identity.
- `overlap`: Iterate over those regions that overlap with a given query region.
- `set_annotation`: Replace annotation values of a region present in the collection.
- `set_score`: Set numeric score values for a region store that represents a real-valued function over the genome.

A region store implementation may support one, several or (in principle) all of the possible interface methods. In general, only methods that can be implemented efficiently are supported. For example, file readers such as the BED parser do not support random-access queries onto files, but indexed structures such as the SQL database support this method. Region store implementations are divided into categories based on which methods they support:

- File readers support only the iteration method. They iterate over each region defined in an input file.
- File writers support only the writing (i.e., adding or appending) method. A region added to the region collection is written to an output file.
- Region databases are in-memory or on-disk stores that support several or all methods. They are used for multi-pass algorithms that need to keep intermediate results in storage, and they provide index structures for efficient processing.
- Region filters transform region streams provided by other region stores by omitting selected regions or modifying region attributes. They support the iteration method.

**Workflows** Region stores can be flexibly combined into *ad hoc* workflows that compute composite operations on several region collections. A simple workflow would consist of a BAM reader combined to a BED writer, i.e., regions produced by the BAM iterator are written (added) to the BED region store. This implements BAM to BED file conversion. As a more complex example, the set difference operation $A \setminus B$ is implemented by (1) constructing region readers for $A$ and $B$, (2) initializing an empty region database for temporary storage, (3) adding all regions from $A$ to database, (4) removing all regions from $B$ from database and (5) writing database contents into output file. For convenience, this composite operation and other similar operations are available in the scripting language APIs.

## 2.4 Region scores and aggregate functions

When regions are interpreted as genomic intervals with numeric score annotations, we can define a real-valued function over the whole genome that we call a *score function*. A set of regions in a data set gives the numeric value of this function at each genomic location. GROK provides functionality to access and manipulate these score functions. Score manipulation (method `set_score`) is done using *aggregate functions*, which define how an updated score is computed from an existing value and an update value. Aggregate functions are listed in Table 1 and their use is illustrated in Section 5.10.

**Example** The following example illustrates a score function over an interval spanning genomic locations 100 to 160 and its modification using the `product` aggregate function. For simplicity, chromosomes and strands are omitted. Before modification,

| Function | Formula | Initial | Description |
|---|---|---|---|
| `count` | $f(x,y) = x+1$ | 0 | Count the number of score updates. |
| `first` | $f(x,y) = x$ | – | Use the first defined value. |
| `last` | $f(x,y) = y$ | 0 | Replace existing value with new one. |
| `min` | $f(x,y) = \min(x,y)$ | – | Use minimum of values. |
| `max` | $f(x,y) = \max(x,y)$ | – | Use maximum of values. |
| `product` | $f(x,y) = xy$ | 1 | Use product of values. |
| `sum` | $f(x,y) = x+y$ | 0 | Use sum of values. |
| `zero` | $f(x,y) = 0$ | 0 | Set score to zero. |

Table 1: Aggregate functions descriptions. Formulas describe how an updated score value is computed from the existing value $x$ and a novel value $y$. Scores are initialized to a value that depends on the aggregate function; the initial value is undefined (–) for some functions. The `last` function defines simple assignment semantics and is the default.

the score function is composed piecewise by three neighboring regions whose score annotation values are 8, 13 and 5. The modification affects the interval from 110 to 150 – denoted as half-open interval $[110, 150)$ – and multiplies all scores with 2. The modified score function is composed of five regions.

```
set_score([110, 150), 2, product)
Before:                 After:
[100, 120): 8           [100, 110): 8
[120, 135): 13          [110, 120): 2*8 = 16
[135, 160): 5           [120, 135): 2*13 = 26
                        [135, 150): 2*5 = 10
                        [150, 160): 5
```

## 3 Region store implementations

GROK provides several implementations of the region store interface for different purposes. Available implementations are described in this section and instructions for using them are in Section 5. Table 2 provides a summary on the methods supported by each region store implementation.

### 3.1 File I/O

GROK supports reading and writing sequencing and genomic data files in multiple formats. All formats are accessed through a similar API that provides a parameter for the format. Format auto-detection is also supported. File formats use the same region model and thus modular format conversion routines can be implemented using GROK as long as the converted formats have similar data content. Format support is as follows:

- BAM, SAM: read & write (also conversion between these formats)
- BAM indexed (BAM+BAI): read (overlap queries supported)
- BED: read & write
- BedGraph: read
- CSV (tab-delimited): read & write
- FASTQ: read & write
- GFF & GTF: read
- VCF: read
- Wiggle: read & write (fixed and variable step)

| Store | `add` | `get` | `iter` | `overlap` | `remove` | `set_annotation` | `set_score` |
|---|---|---|---|---|---|---|---|
| Hash store | y | y | y | – | y | y | – |
| SQLite store | y | y | y | y | y | y | – |
| Redblack store | y | y | y | – | y | y | – |
| Partition store | – | – | y | y | – | – | y |
| IO: readers | – | – | y | – | – | – | – |
| IO: BAM reader | – | – | y | y | – | – | – |
| IO: writers | y | – | – | – | – | – | – |

Table 2: Method support for region store implementations. Partition store refers to a specialized configuration of a red black tree for handling numeric scores for chromosomal intervals.

**CIGAR & PHRED**: BAM, SAM and FASTQ formats contain CIGAR and PHRED quality strings that encapsulate information in individual characters of their content. GROK does not interpret these strings but rather considers them as atomic units. Users wishing to access invidual tokens of information can use libraries such as Biopython to parse these strings.

**CSV**: CSV (tab-delimited) files must have a header row containing column names. GROK autodetects column names for core attributes. Detected names include: chromosome, chr, chrom, seqid, seqname, refseq, start, chromstart, end, chromend, length, strand, id, name. Other columns are considered annotations; an arbitrary number of annotations are supported.

## 3.2 In-memory indexed databases

In-memory databases are indexed collections of regions and their annotations. Regions and annotations are packed into memory in an efficient manner. Any number of region annotations are supported. There are three stores of this kind:

- Hash store: regions are stored in a hash table that allows efficient membership queries but order of regions is random (i.e., not genomic order). Supported methods: `add`, `iter`, `get`, `remove` and `set_annotation`.
- Red-black store: like hash store, but regions are sorted in genomic order in a balanced tree. Supports the same methods as hash store. Iteration is in genomic coordinate order.
- Partition store: specially configured red-black structure for region score manipulation. See Section 5.10 for details.

The hash and red-black stores are suitable for general purpose in-memory processing. Selection between these two depends on the preferred order of iteration of result regions: the red-black store is a good default choice. These stores do not support overlap queries (the `overlap` method); the SQLite store should be used for them.

## 3.3 SQLite backend

The SQLite store uses an on-disk or in-memory SQLite database that supports all operations except `set_score`. Use of disk storage allows scaling to large data sets that do not fit into memory. Regions together with their annotations are stored in the database, and SQL indices provide efficient implemention of `get`, `remove` and `overlap` methods. When using the in-memory option or a temporary database file, database contents are deleted when the region store is closed. The SQL interface may add some general overhead compared to pure memory operations, so for small data sets, hash and red-black databases may be more efficient.

## 3.4 Filtering and transforming store

GROK provides a filter store for processing regions provided by other region stores. The filter store takes another region store as a parameter and provides and iterator over the regions in this parent store. The region filter store can be configured to act as a filter, i.e., to selectively output only regions satisfying certain criteria, or as a transformer, i.e., to modify attributes of regions, or both.

The following filtering options are available for core attributes and optional annotations:

- Select regions whose strand matches given value (e.g., only forward strand).
- Select regions whose numeric attribute is between low bound $L$ and high bound $H$. The attribute can be left coordinate, length or any numeric annotation.

The following transformations are available:

- Flip (switch) the strand between forward and reverse. The ”none” pseudo-strand is not affected by flipping.
- Strand flip with a fixed strand: all regions are assigned the same given strand.
- Shift the region along the chromosome, modifying left and right positions but keeping length constant.
- Expand or shrink the region interval. Left and right positions as well as length can be modified.

## 3.5 Merging store

The merging store is similar to the filtering store but provides a specialized merge transformation that combines adjacent regions into one. Region *adjacency* is defined by the parent region store iterator: regions are adjacent if the iterator outputs them one after another. Only regions in the same chromosome and strand are merged. For non-stranded merging, first transform the strands to having the same value with the filtering store.

The merging store has a parameter for the maximum gap between adjacent regions. The gap is defined as the number of nucleotides between two regions that would be needed to connect the regions. If the gap is larger than this, the regions are not merged and are provided as separate regions in the iterator. For regions having a numeric score annotation, the store also has a parameter for maximum score difference.

# 4 Installation and requirements

Choosing the GROK distribution and installation method depends on the intended use of the library, in particular the scripting language bindings the user plans to use. All GROK distributions include bundled versions of SQLite 3 and SAMtools so these dependencies do not need to be explicitly installed. Currently, only Unix-based operating systems are supported.

## 4.1 Unix

All Unix installation methods require a C++ compiler (e.g., gcc 4.1 or later), make and zlib.

### 4.1.1 Source package

Compiling:

1. Download the source package `grok-x.y.z.tar.gz`
2. Unpack with `tar zxvf grok-x.y.z.tar.gz`
3. Run `./configure`
4. Compile C++ sources and a static GROK library with `make`. The library `src/libgrok.a` includes SQLite and SAMtools.
5. (Optional) Build C++ API documentation (Doxygen) with `make doxygen`
6. (Optional) Execute unit test cases with `make unittest`

### 4.1.2 R bindings

Requirements: R 2.12+, R development files (Debian/Ubuntu package `r-base-dev`)

Installation:

1. Download the R package `grok_x.y.z.tar.gz`
2. Install by running `R CMD INSTALL grok_x.y.z.tar.gz` (may need administrator privileges)

### 4.1.3 Python bindings

Requirements: Python 2.x, Python development files (Debian/Ubuntu package `python-dev`)

Installation:

1. Download the source package `grok-x.y.z.tar.gz`
2. Unpack with `tar zxvf grok-x.y.z.tar.gz`
3. Run `./configure`
4. Build bindings with `make python`
5. Install to Python repository with `make python-install` (may need administrator privileges)
6. (Optional) Execute test cases with `make python-test`

The `locate` tool must be able to find the Python include files, so if you recently installed `python-dev`, you may need to run `updatedb` before `configure`.

#### 4.1.4 Lua bindings

Requirements: Lua 5.1, Lua development files (Debian/Ubuntu package `liblua5.1-0-dev`)

Installation:

1. Download the source package `grok-x.y.z.tar.gz`
2. Unpack with `tar zxvf grok-x.y.z.tar.gz`
3. Run `./configure`
4. Build bindings with `make lua`
5. Install to Lua repository by manually copying `lua/grok.so` to your Lua C search path (e.g., `/usr/local/lib/lua/5.1/`); may need administrator privileges. Lua C search path can be shown with `lua -e 'print(package.cpath)'`
6. (Optional) Execute test cases with `make lua-test`

The `pkg-config` tool must be able to find the library definition file for Lua. This should be provided by the Lua-dev package.

# 5 Using the R, Python and Lua APIs

GROK provides language bindings for the scripting languages R, Python and Lua. The scripting interface enables implementing complex sequencing operations using a high level language. All core GROK operations can be accessed from scripting languages. The APIs for all supported scripting languages are similar to each other, with only minor differences to accommodate for differences in the languages. The code for language bindings is generated with SWIG (`http://swig.org/`), which ensures a consistent API. This section mostly uses R as the illustration language. See section 5.11 for a detailed API reference and calling conventions for Python and Lua.

## 5.1 Basic usage

In R, the GROK library is imported with:

```
require(grok)
# GROK methods are named grok.METHOD
```

In Python, the corresponding statement is:

```
import grok
# GROK methods are accessed as grok.METHOD
```

In Lua:

```
require("grok")
-- GROK methods are accessed as grok.METHOD
```

## 5.2 Data types

The scripting API is object oriented and closely follows the concepts in Sections 2 and 3. Two central classes in the API are `RegionStore`, which corresponds to a collection of DNA regions, and `Region`, which represents a single region. Objects of `Region` provide access to region attributes, including core attributes and annotations. They are represented as key–value data structures using the facilities of the scripting language. In R, `Region` objects are R lists; in Python, they are Python dictionaries; and in Lua, they are Lua tables. Objects of other classes are opaque types that are accessed using their methods. A supplementary class in the API is `RegionIterator`, which is used for iterating over a region store.

## 5.3 Reading and writing regions in R, Python and Lua

The following example illustrates basic region I/O using R. The code instantiates a BAM file reader using `grok.reader` and a BED writer using the `grok.writer` method, and writes all regions using the `grok.add` method. This implements a BAM-to-BED file converter.

```
require(grok)
input <- grok.reader("input.bam")
output <- grok.writer(input, "output.bed")# Inherit annotations from input
grok.add(output, input)
```

The `grok.reader` and `grok.writer` functions return `RegionStore` objects that provide methods for accessing the regions. Both functions have an optional parameter for file format, which is by default auto-detected.

The region writer function takes optional `RegionStore` objects as argument that determine which region annotation are supported. A writer supports the region annotations of the region stores given as argument to the constructing function.

In Python, the corresponding code is as follows. Notice the use of method calls for the `input` and `output` objects.

```
import grok
input = grok.reader("input.bam")
output = input.writer("output.bed")
output.add(input)
```

Finally, the Lua code is similar to Python, the main difference being the use of the Lua syntax for method calls:

```
require("grok")
input = grok.reader("input.bam")
output = input:writer("output.bed")
output:add(input)
```

## 5.4 Iterating over regions

Region stores provide iterators that are used to access individual regions in the store. Iterators are supported by all region store implementations with the exception of writers. Multiple iterators can be instantiated for a single region store, except for filtering, transforming and merging stores that only provide a single iterator. The following example iterates over all records of a BAM file.

```
require(grok)
input <- grok.reader("input.bam")
iter <- grok.iter(input)
region <- grok.next(iter)
while (is.list(region)) {
  cat(sprintf("Region: chromosome %s, strand %d, left %d, length %d\n",
    region$seqid, region$strand, region$left, region$length))
  region = grok.next(iter)
}
```

An iterator is created with the `grok.iter` method. The iterator is advanced with `grok.next`, which returns the next region or `FALSE` if all regions have been iterated over.

## 5.5 Using region databases

Region databases are in-memory or on-disk structures that hold regions. They enable writing multi-pass processing algorithms and provide index structures for efficient queries. This section considers regions as identity tuple (see 2.2) and the operations are traditional database operations (appending and removing). For operations corresponding to chromosomal interval interpretation, see section 5.10.

Region databases are instantiated with global constructor functions that take zero or more existing region stores as argument. These define which annotations are supported by the database. If no arguments are given, the database does not store optional annotations. Otherwise, the set of annotations is the union of annotations in given stores.

The following example creates a union collection that contains all regions and annotations present in two input BED files. The union is stored in an SQLite database. Notice that the `add` method returns the original region store so multiple `add` calls can be chained.

```
require(grok)
input1 <- grok.reader("input1.bed")
input2 <- grok.reader("input2.bed")
db <- grok.sqlite.store(input1, input2) # Create database
grok.add(grok.add(db, input1), input2)
writer <- grok.writer(db, "union.bed")
grok.add(writer, db) # Write result
```

Other database constructors are `grok.hash.store`, and `grok.reblack.store`. Their constructors have a similar interface.

The `grok.add` method also accepts region iterators and individual regions in addition to region store objects. The following example uses the hash store:

```
require(grok)
input <- grok.reader("input.bed")
db <- grok.hash.store(input)
grok.add(db, grok.iter(input)) # Add regions using iterator
region <- list(seqid="chr1", left=100, length=50, strand=1)
grok.add(db, region) # Add single region
```

The `remove` method removes regions from a database based on region identity tuples. The following example keeps regions that are present in `input1.bed` but not in `input2.bed` (set difference). Like `add`, `remove` also accepts iterators and individual regions in addition to region stores.

```
require(grok)
input1 <- grok.reader("input1.bed")
input2 <- grok.reader("input2.bed")
db <- grok.sqlite.store(input1, input2)
grok.add(db, input1)
grok.remove(db, input2)
```

### 5.5.1 Sorting regions in genomic order

The redblack store can be used to sort regions in genomic order. This occurs automatically when regions are added to the redblack data structure, since the store is structurally always sorted. The following example places all regions from a BED file into a redblack store and creates an iterator that produces regions in genomic order.

```
require(grok)
input <- grok.reader("input.bed")
db <- grok.redblack.store(input)
grok.add(db, input)
iter <- grok.iter(db) # iter now provides sorted regions
```

## 5.6 Overlap queries

An overlap query provides those regions in a region store that share one or more nucleotide positions with a query region. This functionality can be used to fetch sequencing reads that overlap with a gene, for example. The overlap operation is a filter that produces a subset of a region collection: the overlapping regions are not modified by the filter. In order for two regions to overlap, they must be in the same chromosome and strand. If non-stranded operation is required, the strands should first be transformed using `grok.flip` (see Section 5.8).

Overlap queries require explicit support from the underlying region store for efficiency reasons. The SQLite database provides this support for regions from any origin (e.g., BED files). In addition, BAM reader has direct support for overlap queries if BAM index files (BAI) are used.

The following example filters those regions from a BED file that overlap with the genomic coordinates $[1000, 1500)$ in the reverse strand of chromosome 1 and writes these regions to an output file. The query region is represented as an R list. The object returned by `grok.overlap` is a regular region iterator that can also be iterated over in a normal manner (see Section 5.4).

```r
require(grok)
input <- grok.reader("input.bed")
db <- grok.sqlite.store(input)
grok.add(db, input)
query <- list(seqid="chr1", left=1000, length=500, strand=-1)
iter <- grok.overlap(db, query)
output <- grok.writer(db, "output.bed")
grok.add(output, iter)
```

The following example shows a similar filtering operation for a BAM+BAI file. In this case, an extra region database is not necessary as the file format supports overlap queries directly.

```r
require(grok)
input <- grok.reader("input.bam") # input.bam.bai must exist
query <- list(seqid="chr20", left=1000, length=500, strand=-1)
iter <- grok.overlap(input, query)
output <- grok.writer(input, "output.bam")
grok.add(output, iter)
```

A common need is to execute an overlap query in a loop, producing those regions in store A that overlap with any region in store B. This can be implemented using a variant of `grok.overlap` that takes a store as argument. There are two modes, termed forward and backward, of such queries with different performance charasteristics. See Section 5.11.5 for details. Below is an example of a reverse overlap query that returns those regions in `input1.bed` that overlap with regions in `input2.bed`. We first create a partition store and fill it with regions from `input2.bed`. Then we execute a reverse overlap query using `input1` and the partition store.

```
require(grok)
input1 <- grok.reader("input1.bed")
input2 <- grok.reader("input2.bed")
db <- grok.partition.store()
grok.set.score(db, input2)
result <- grok.overlap(input1, db, FALSE)
output <- grok.writer(db, "output.bed")
grok.add(output, result)
```

## 5.7 Set operations: union, intersection and difference

Set operations are used to compare and combine several sequencing data sets (region stores) together. Union $A \cup B$ includes all regions or genomic locations that are present in at least one store, while intersection $A \cap B$ requires presence in all stores. Set difference $A \setminus B$ is an asymmetric operation that removes regions or locations in $B$ and $A$. In addition, GROK provides a generalization of union and intersection, `freq`, that takes low and high frequency bounds as parameter and filters regions based on their occurrence frequency. This implements a k-out-of-n operation. Union corresponds to `freq(1, n)` and intersection to `freq(n, n)`.

All set operations can be computed using standard region store methods `grok.add`, `grok.remove` and `grok.set.score`, so the operations discussed here are convenience wrappers over these methods. For example, a union can be computed manually with the `grok.add` method.

There are two variants of all set operations that correspond to relation and interval interpretations of regions (see Section 2.2). Relation methods are `grok.union`, `grok.intersection`, `grok.diff` and `grok.freq`. Interval methods are `grok.unionL`, `grok.intersectionL`, `grok.diffL` and `grok.freqL`; the postfix `L` stands for location.

Relation set operations (with no `L`) consider two regions equal if their identity tuples are equal, so it's possible to have two regions covering the same genomic location in a union result. In contrast, interval set operations (with `L`) have each genomic location only once in a union. The `L` variants use `grok.set.score` for computing results, while the relation variants use `grok.add` and `grok.remove`.

The following example illustrates the use of chained union and difference operations with three BED files. The result of the computation is the unique regions (`grok.unique`) or unique genomic locations (`grok.uniqueL`). The `grok.unique` result region store contains regions from input1.bed whose identity tuple does not match any regions from input2.bed or input3.bed. In contrast, `grok.uniqueL` contains regions corresponding to genomic intervals that are present in input1.bed but not in input2.bed or input3.bed.

```
require(grok)
input1 <- grok.reader("input1.bed")
input2 <- grok.reader("input2.bed")
input3 <- grok.reader("input3.bed")
unique <- grok.diff(input1, grok.union(input2, input3))
uniqueL <- grok.diffL(input1, grok.unionL(input2, input3))
```

## 5.8 Transforming regions

Most attributes of regions can be modified using transformation operations. Core attributes such as strand and genomic position can be modified by iteration for all region stores that support the iteration operation, i.e., all region collections except writers. This does not modify region stores in place. Optional annotations can be modified for region databases (see Section 5.9). Naturally, it is also possible to iterate over regions and manually manipulate the region objects (lists).

The strand of regions is modified with the `grok.flip` operation. There are two variants of this function: one flips strands from forward to reverse and vice versa, and the other sets all strands to a constant value. These operations produce a region store that can be iterated over only once. In the following example, `flipped` is a region store that contains the regions of the BED files with their strands flipped. In contrast, `reverse` contains the regions whose strands are all reverse (-1).

```
require(grok)
input <- grok.reader("input.bed")
flipped <- grok.flip(input)
reverse <- grok.flip(input, -1)
```

The length and position of regions are modified with `expand` and `shift`. The former modifies both the location and length of regions, while the latter keeps the length constant but modified the location. These operations are strand aware: they define a ”positive direction” that is towards the end of sequence for forward strand and start of sequence for the reverse strand. Like `grok.flip`, these operations also produce a one-time iterable store.

In the following example, all regions are first expanded 100 nucleotides from their start position and shrunk 20 nucleotides from their end position. Then, these transformed regions are shifted 10 nucleotides in the positive direction. Start position is the leftmost position for forward strand and the rightmost position for reverse strand. For example, a reverse strand interval covering bases from 500 to 700 – denoted [500, 700) – is first expanded/shrunk into [520, 800) and then shifted into [510, 790).

```
require(grok)
input <- grok.reader("input.bed")
transformed <- grok.expand(input, 100, -20)
transformed <- grok.shift(transformed, 10)
```

## 5.9 Modifying region annotations

Database stores support modifying region annotations. Annotations are modified using `set_annotation`. This modifies the region store in place. In the example below, BED regions are loaded into an in-memory database with a hash index. The numeric score annotation of all regions is multiplied by 2 and the result is saved in the database. In addition to modifying numeric annotations, string-valued annotations can also be assigned using `set_annotation`.

```
require(grok)
input <- grok.reader("input.bed")
db <- grok.hash.store(input)
grok.add(db, input)
grok.set.annotation(db, db, "score", 2, "product")
```

The `set_annotation` method receives five arguments, of which the last is optional. The first is the region store to be modified. The second specifies the region collection that is affected by the operation. In the example, all regions in the database are updated, but it is also possible to update only a subset. The third argument specifies the annotation name, Score, which corresponds to the numeric scores in BED files. The fourth gives the value that replaces existing score annotations. How this value is interpreted depends on the last argument, which specifies an aggregate function. This function that computes a new annotation value from the existing and new value. In this example, the `product` function is used to compute new scores with the formula $f(x) = 2x$. The default aggregate function, `last`, simply assigns the given value and discards the old value. Aggregate functions are listed in Section 2.4.

An important difference between `set_annotation` and `set_score` (Section 5.10) is that the former considers regions as relations and only modifies their attributes but does not split regions. In contrast, `set_score` considers the region store as a set of non-overlapping intervals and may split regions when modifying scores. Another difference is that `set_score` works only with numeric annotations while `set_annotation` also supports strings.

## 5.10 Working with numeric region scores

Region score functions (see Section 2.4) are transformed with algebraic operations using the `set_score` method. This method is only supported by the red black stored configured as a *partition*, which implies that the store does not contain any overlapping regions. A partition is constructed with the `grok.reblack.partition` constructor.

In the following example, the maximum scores of two Wiggle files are computed using a red black partition. The red black store is first constructed empty, with a zero score in all genomic locations. The first `set_score` call sets scores on those locations present in the first Wiggle file to the value given in the file. The second call sets scores to the maximum of the scores in input files using the aggregate function `max`. In the result score function, the value at position $i$ is $f(i) = \max(W_1(i), W_2(i), 0)$, where $W_1$ and $W_2$ are the two Wiggle files. Result scores are written to an output Wiggle file.

```
require(grok)
input1 <- grok.reader("input1.wig")
input2 <- grok.reader("input2.wig")
partition <- grok.partition.store()
grok.set.score(partition, input1) # default aggregate function "last"
grok.set.score(partition, input2, "max")
writer <- grok.writer(partition, "output.wig")
grok.add(writer, partition)
```

#### 5.10.1 Counting short reads along genome

The following example computes short read counts for each genomic location based on a BAM file. This is done using the max aggregate function that sets scores to the number of regions matching each location.

```
require(grok)
input <- grok.reader("input.bam")
partition <- grok.partition.store()
grok.set.score(partition, input, "count")
writer <- grok.writer(partition, "output.wig")
grok.add(writer, partition)
```

### 5.11 Method reference

This section provides a reference of all methods and functions supported by GROK. Method and class names are similar in supported scripting languages, with some minor modifications to accomodate language differences. Methods are documented in a programming language independent format. Method signatures are formatted as:

`ReturnType Class::method_name(Type1 name1, Type2 name2=default)`

Variable number of arguments are formatted as `Type... name`. Functions without a class are global non-method functions.

For documentation purposes we define RegionSource as a base class of RegionIterators and RegionStores and where it is used both types are accepted. In the variable case any combination of these types is accepted. However the variable argument amount is limited to 10 for technical reasons.

In R, all methods are accessed using globally defined functions that take the ”self” object as first parameter. All R function names are prefixed with `grok.` (for region store). To accommodate for R conventions, function names use dots (.) instead of underscores (_). For example. the method with signature

```
RegionStore RegionStore::remove_annotation(string annotation)
```

is defined in R as:

```
grok.remove.annotation <- function(self, annotation) { ... }
```

In Python, regular method call syntax `obj.method(args)` is used and function names have underscores. In Lua, the method call syntax is `obj:method(arg)` and underscores are used in names.

### 5.11.1 Region store constructors

```
RegionStore reader(string filename, string format="auto")
```

Create a region reader over a file. The `format` parameter gives the file format and is one of `auto` (use autodetection), `bam`, `bed`, `bedgraph`, `csv`, `fastq`, `gff`, `gtf`, `sam`, `vcf`, `wiggle`. The regions can be iterated over multiple times.

```
RegionStore RegionSource::writer(string filename, string format="auto")
```

Create a region file writer from the current region source. File name and format are specified in a similar manner as for region readers. For the Wiggle format, an additional format `wigglevariable` is supported that produces variable step output. The writer uses region annotation definitions from the current region store, although most file formats support only a limited set of annotations.

```
RegionStore RegionStore::hash_store(RegionSource... sources)
RegionStore RegionStore::redblack_store(RegionSource... sources)
RegionStore RegionStore::sqlite_store(RegionSource... sources)
RegionStore hash_store(RegionSource... sources)
RegionStore redblack_store(RegionSource... sources)
RegionStore sqlite_store(RegionSource... sources)
RegionStore partition_store()
```

Instantiate a new empty region database. There are constructors for all four supported database implementations; they can be accessed using a method or a static function. The only difference is that in the method form, the current region store is implicitly added to the list of `sources` arguments. These variable length arguments are used to define the set of supported region annotations in the new store. The union of all annotation definitions is used. Constructors do not add any regions to the new store. The partition store does not take arguments and thus has only a static constructor. **Note**: RegionIterator objects can also be given as argument for all constructors in addition to RegionStore objects.

### 5.11.2 Region store accessors

```
RegionStore RegionStore::add(RegionSource source)
RegionStore RegionStore::add(Region source)
```

These methods insert (or write) regions from given source to current region store. The source can either be another region set and all its regions, or an iterator over a region collection, or a single region. The methods return the store itself to enable method chaining. Insertion semantics depend on the region store. For I/O writers, the method writes the regions to the output file. For databases, the regions are added to the collection using region relation semantics. Databases ensure that no duplicates are present: if a duplicate region is added, the existing one is replaced with the new one.

```
RegionStore RegionStore::remove(RegionSource source)
RegionStore RegionStore::remove(Region source)
```

Remove regions present in given source from the current store. These methods are only supported for region databases. Removal is done using region identity, i.e., a region is removed from the current region if a region with an equal identity tuple is present in the source. Like `add` methods, the source can be a store, iterator or single region. The methods return self.

```
RegionStore RegionStore::get(Region reg)
```

Fetch a specific region from current region store. The region is searched using identity tuples. If a corresponding region is found, it is returned; otherwise, a null value is returned.

```
RegionStore RegionStore::set_annotation(RegionSource source,
     string annotation, object value, strint aggregate="last")
RegionStore RegionStore::set_annotation(Region source,
     string annotation, object value, strint aggregate="last")
```

Assign region annotations in the current region store. Source regions define the affected set of regions, which is a subset of all regions in the current store. Assignment can affect all, some or none of the regions in the current store, depending on the regions present in `source`. Source regions are associated to regions in the current store using region identity tuples. Annotation name and the new value are given with `annotation` and `value`. Value can be a string, integer and float depending on the annotation type. For numeric annotations, an aggregate function (`aggregate`) can be given that defines how a new annotation value is derived from the existing and new ones (see Section 2.4). The SQLite store only supports the `last` and `counter` aggregate functions.

---

```
RegionStore RegionStore::remove_annotation(string annotation)
RegionStore RegionIterator::remove_annotation(string annotation)
```

Remove the given annotation definition from the current store. The annotation is given by name. Return self.

### 5.11.3 Set operations

---

```
RegionStore union(RegionSource... sources)
RegionStore unionL(RegionSource... sources, String aggregate="last")
```

Compute union by returning a newly created store that contains all regions (`union`) or all genomic locations (`unionL`) present in any argument sources without duplicates. Equal to `freq(1, n, sources)` or `freqL(1, n, sources)`, where `n` is the number of arguments. Note: the `aggregate` argument must be given with a keyword syntax (`aggregate="X"`) and it is not available in Lua.

---

```
RegionStore intersection(RegionSource... sources)
RegionStore intersectionL(RegionSource... sources, String aggregate="last")
```

Compute intersection by returning a newly created store that contains regions (`intersection`) or genomic locations (`intersectionL`) present in all argument sources. Equal to `freq(n, n, sources)` or `freqL(n, n, sources)`, where `n` is the number of arguments. Note: the `aggregate` argument must be given with a keyword syntax (`aggregate="X"`) and it is not available in Lua.

---

```
RegionStore diff(RegionSource source1, RegionSource source2)
RegionStore diffL(RegionSource source1, RegionSource source2)
```

Compute asymmetric set difference by returning a newly created store that contains regions (`diff`) or genomic locations (`diffL`) that are present in `sources1` but not in `sources2`.

---

```
RegionStore freq(int low, int high, RegionSource... sources)
RegionStore freqL(int low, int high, RegionSource... sources, String aggregate="last")
```

Return a newly created store that contains regions (`freq`) or genomic locations (`freqL`) that are present in at least `low` and at most `high` argument sources. The count of each region (`freq`) or genomic location (`freqL`) is computed by iterating over all given region sources. If a store contains duplicate regions, those regions are counted several times for the store. Duplicates can be removed by first wrapping the region collection is a hash store or red black partition. Note: the `aggregate` argument must be given with a keyword syntax (`aggregate="X"`) and it is not available in Lua.

### 5.11.4 Region transformers

---

```
RegionStore RegionSource::expand(int start, int end)
RegionStore RegionSource::shift(int length)
```

Modify the genomic coordinates of each region in the current store. The `expand` method modifies both start and end locations independently. Positive arguments expand regions and negative arguments shrink them. The method is strand aware so that the start position is the rightmost position for the negative strand. The `shift` method moves regions in left or right directions and keeps their length constant. For positive length, forward strand regions are moved to the right and reverse strand regions to the left; negative length behaves in the opposite direction. The methods return a new region store that can be iterated over at most once.

---

```
RegionStore RegionSource::flip()
RegionStore RegionSource::flip(int fixed)
```

Change the strand of each region in the current store. In the first variant, the strand is flipped from forward to reverse ($1 \rightarrow -1$) and vice versa ($-1 \rightarrow 1$); the none pseudo-strand (0) is not affected. In the second variant, all regions are assigned a fixed strand given as parameter (-1, 0 or 1).

---

```
RegionStore RegionSource::merge(int max_gap, int max_score_diff)
```

Merge adjacent regions in the current region store. Regions are adjacent when they are produces after each other in iteration. Merging combines two or more regions into one region that spans all coordinates of the merged regions. The `max_gap` parameter specifies the maximum allowed gap in nucleotides between regions for merging. The `max_score_diff` argument specifies the maximum allowed absolute difference in the scores between regions for merging. It is only used if the regions have a score annotation.

---

`RegionStore RegionSource::filter(String attribute, double low, double high=DBL_MAX)`

Filter regions by values of a numeric attribute. The resulting store produces regions whose attribute value is at least `low` and at most `high`. The attribute may be one of the special values `strand`, `tag` (identity tag) or `length` (region length); or the name of a numeric annotation. For `strand`, the `high` parameter is not used and `low` is used for equality comparison against the strand (-1, 0 or 1). If `high` is omitted, a maximum value (no upper bound) is used.

### 5.11.5 Region iterators and overlap queries

---

`RegionIterator RegionStore::iter()`

Create a new iterator over all regions in the current region store.

---

`RegionIterator RegionStore::overlap(Region query)`

Create a new iterator over those regions in the current region store that overlap with the query region. Regions overlap if they are in the same chromosome and strand and have at least one nucleotide coordinate in common. The overlap iterator is accessed like a regular iterator. See Section 5.6 on how to define a query region representation in the scripting language.

---

`RegionIterator RegionStore::overlap(RegionStore query, boolean forward=true)`

Create a new iterator over regions that overlap with any region in the query store. There are two modes in the method, forward and backward, controlled by the forward flag.

In the forward mode, we iterate over each region in the query and return every region in the current store that overlaps with the query region. This calls overlap(Region) repeatedly.

In the reverse mode, we iterate over regions in the current store and check for each whether it overlaps with any region in the query store. In this case, the query store must support the overlap(Region) method.

Both modes produce regions in the current store as result; the forward mode may return the same region multiple times, whereas the reverse mode returns each region at most once.

---

```
Region RegionIterator::next()
```

Produce the next region in the iterator, or return a ”null” value if iteration has finished. If R, `FALSE` is returned after iteration. In Python, `None` is returned. In Lua, `nil` is returned.

---

```
void RegionStore::close()
void RegionIterator::close()
```

Explicitly close the store or iterator. Closing is also done automatically during garbage collection. For writer stores this also includes flushing the file.

---

```
void RegionIterator::remove()
```

Remove the currently active region from the region set over which iteration is done. The currently active region is the latest region produced by `next()`. Not all iterators support removing.

### 5.11.6 Region score manipulation

---

```
RegionStore RegionStore::set_score(RegionStore source,
     string aggregate="last", string annotation="")
RegionStore RegionStore::set_score(RegionIterator source,
     string aggregate="last", string annotation="")
RegionStore RegionStore::set_score(Region source,
     string aggregate="last", string annotation="")
```

Manipulate numeric scores of genomic intervals. In the context of these methods, regions are interpreted as non-overlapping intervals that do not have intrisic identity. New scores are assigned as follows. For each region in the source, the corresponding interval from the current region store is queried and the scores of these two regions are combined using an aggregate function (given by `aggregate`; see Section 2.4). A source region may correspond to multiple regions in the current store. The name of the score annotation is given with `annotation`: if the string is given, the default score annotation is used. The methods return self.

# 6 Using the command line interface

The CLI offers easy access to common operations including file conversion, set operation and overlap queries in a single utility called `grok`, implemented in Python. Thus many conventions are the same as in the scripting APIs. The utility is located in `cli/grok` is the distribution: copy this file to your executable path to install it system-wide. The CLI requires the installation of Python bindings for GROK first.

This manual describes first the general usage and then documents each command individually.

## 6.1 General usage

Simply typing `grok` on the command line will show a synopsis for each command and list what the flags do. Otherwise the grok utility always needs a subcommand to do something useful. Option flags may be specified at any location on the command line and will not affect parsing of positional parameters.

A specific operation is chosen with the first positional parameter and further positional parameters are directed to it. Option flags may affect all commands or only apply to specific commands. The globally applicable flags are described in this section and others are explained in the applicable subsections.

The general synopsis is

```
grok [options] <command> <extra parameters> [input file] ... [output file]
```

The mandatory extra parameters depend on the command and for most commands there are no extra parameters. The amount of accepted input files also varies. However there is always only one output file, the result set.

File types are deduced automatically from filenames for both input and output files but may be stated explicitly with the -t and -T flags respectively.

## 6.2 File conversion

To perform file conversion the command to use is ”convert”. It needs one input file and writes the regions in that file to an output file.

**Example** The following example shows how to convert a BAM file to a SAM file.

```
grok convert input.bam output.sam
```

## 6.3 Set operations

Set operations take multiple input files in any of the available formats and produce an output file in the specified format. The commands are ”union”, ”intersection”, ”freq” and ”difference”. There are two variants of these commands corresponding to “relation” or “interval” (location) interpretations of genomic regions. The reglation variant is the default and the location variants are selected with the `-l` flag. For the location variants, the aggregate function for specifying score annotation computation can be assigned with the `-a` flag.

They commands do not take extra parameters and allow an arbitrary amount of input files except for the following exceptions: The ”freq” command needs two extra parameters i.e. the minimum and maximum counts for inclusion in the result. The ”difference” command takes exactly two input stores.

**Example** The following example shows some set operations.

```
# Compute short read coverage
grok union -l -a count input.bam output.bed

# Remove genesB from genesA by identity
grok difference genesA.gff genesB.bed output.bed
```

# 7 Using the C++ API

The C++ API provides low level access to GROK internals. It can be utilized in two ways: as a library in a user application to write custom analysis algorithms in C++, or to extend GROK itself with new region store implementations. The C++ API is accompanied with a SWIG interface that binds GROK to scripting languages. This interface is used to generate bindings for languages supported by SWIG.

Full details of the object oriented C++ API are in Doxygen generated documentation; this section provides an overview of available classes and functions. Key classes are shown in Figure 1. The two most relevant data structures are `Region`, which represents a single region, and `RegionStore`, which defines the interface for region stores. Concrete region store implementations extend the latter by providing implementations of virtual methods.

Auxiliary classes include `SequenceSet` and `Registry` that provide mappings between numeric and string chromosome and annotation names. For efficiency reasons, chromosomes and annotations are referred to using integers in `Region`; these two classes provide interpretation for the integer values.

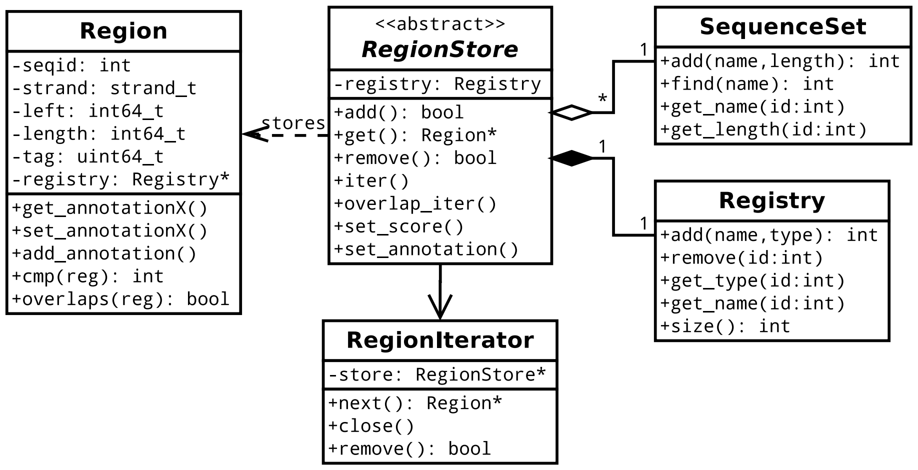

Figure 1: UML class diagram showing an overview of the C++ API.

## 7.1 Accessing region attributes

Individual regions are represented by the `Region` class. This class has accessors for core attributes and annotation values. Core attributes are accessed with `{get,set}_A` methods, where `A` is one of `seqid`, `strand`, `left`, `length`, `right` or `tag`. Coordinates are 0-based. Tags can be created by hashing string identifiers with `set_tag(string)`. Regions are compared using `cmp` that takes into account the four core fields to define relationships between regions. The method `overlaps` tells whether two regions overlap with each other.

The sequence ID links the region to a corresponding sequence such as a chromosome. The ID is an integer that is mapped to a sequence name by an instance of `SequenceSet`. For example, the sequence name corresponding to ID 4 may be `chr5`. For memory efficiency, `Region` does not store a reference to a `SequenceSet`; this reference is available in the region store that houses the region.

Additional annotations are stored in an array-like structure containing integers, floating points and strings. Annotations are accessed using `get_annotationX` and `set_annotationX` where `X` denotes annotation type (`I` = integer, `R` = real, `S` = string). Properties of annotations, such as name, are defined in an annotation registry (`Registry`). A pointer to the registry is stored in `Region` objects. Related regions belonging to the same region store share the same registry, so registering new annotations must be done before the `Region` objects are created. Adding annotations renders existing `Region` objects invalid unless done through `Region::add_annotation`.

**Example** The following example creates a region object and auxiliary objects (sequence set and registry). The region supports one real-value annotation Score, whose value is assigned a (rough) approximation of $\pi$.

```
#include <iostream>
#include "Region.hpp"
#include "Registry.hpp"
#include "SequenceSet.hpp"
int main() {
    SequenceSet &seqset = global_seqset;
    Registry registry;
    int seqid = seqset.add("chr1");
    int annotation_id = registry.add("Score", ANN_REAL);
    Region region(seqid, STRAND_FORWARD, 100, 10, 0xDEADBEEF, &registry);
    region.set_annotationR(annotation_id, 3.14159);
    std::cout << seqset << registry << region;
    return 0;
}
```

When executed, the program prints the following:

```
Sequence set:
  ID: 0, name: , length: 0
  ID: 1, name: chr1, length: 0
Registry:
  ID: 0, name: Score, type: real
Region:
  Sequence ID: 1
  Strand: FORWARD
  Span: [100, 110)
  Length: 10
  Tag: 0xdeadbeef
  Score: 3.14159
```

## 7.2 Using region stores

Region stores are collections of regions represented by `RegionStore` instances. Available region store implementations are show in Table 3. Region stores have methods for iterating, adding and removing regions and other functionality. Region store implementations generally do not support all methods: unsupported methods throw `not_supported` exception. Region stores are stored as shared pointers that free memory automatically; a typedef `RegionStoreP` is defined for convenience. As a convention, region store classes have a static `create` method to construct the region store wrapped in `shared_ptr`.

| Store | Class |
|---|---|
| Generic reader | `FileReader` |
| BAM/SAM reader | `BAMReader` |
| BAM/SAM writer | `BAMWriter` |
| BED reader | `BEDReader` |
| BED writer | `BEDWriter` |
| BedGraph reader | `BEDGraphReader` |
| CSV reader | `CSVRegionReader` |
| CSV writer | `CSVRegionWriter` |
| FASTQ reader | `FASTQReader` |
| GFF reader | `GFFReader` |
| VCF reader | `VCFReader` |
| Wiggle reader | `WiggleReader` |
| Wiggle writer | `WiggleWriter` |
| Merge | `MergeStore` |
| Filter | `TransformStore` |
| Redblack store | `RedblackStore` |
| Redblack partition | `RedblackPartition` |
| Hash store | `HashStore` |
| SQLite store | `SQLiteStore` |

Table 3: Region store implementations. File readers can be accessed using the general purpose FileReader, although using classes for individual formats is also possible.

### 7.2.1 Iterating over regions and I/O

Iteration over regions is done with the `iter` and `overlap_iter` methods. This method returns a pointer to an iterator (`RegionIterator`) that provides methods for advancing (`next)` and closing the iterator (`close`). The `close` method must be called to dispose the iterator; the iterator should not be manually delete'd.

The following example iterates over all entries of a BAM file and writes them to a BED file. The BAM reader is accessed through `FileReader` which also provides file format autodetection. A BED writer is created with an explicit call to the corresponding constructor. The BAM reader object is provides as a ”model” for the BED writer in order to transfer sequence set and annotation registry contents.

```
#include <iostream>
#include "io/bed.hpp"
#include "io/FileReader.hpp"
int main() {
    RegionStoreP reader(FileReader::create("input.bam"));
    RegionStoreP writer(BEDWriter::create("output.bed", &*reader));
    RegionIterator *iter = reader->iter();
    const Region* region;
    while ((region = iter->next()) != NULL) {
        writer->add(*region);
    }
    iter->close();
    return 0;
}
```

### 7.2.2 Using region databases

Region databases are initialized as empty with one of the constructors in Table 3 and are accessed and modified using `add`, `remove`, `get` and `set_annotation` methods. The following example constructs an in-memory SQLite store and populates it with the contents of a file. The SQLite store supports the same annotations as the file reader; annotation definitions are passed to the database in the constructor.

```
#include <iostream>
#include "io/FileReader.hpp"
#include "db/SQLiteStore.hpp"
int main() {
    RegionStoreP reader(FileReader::create("input.bam"));
    RegionStoreP db(SQLiteStore::create(":memory:", &*reader));
    RegionIterator *iter = reader->iter();
    const Region* region;
    while ((region = iter->next()) != NULL) {
        db->add(*region);
    }
    iter->close();
    return 0;
}
```

## 7.3 Implementing new region stores

New implementations for region stores are implemented by deriving a subclass of `RegionStore` and implementing some subset of virtual methods. File readers can alternatively be derived from `StreamReader` or `CSVStreamReader`. If the region store supports iteration, it may need to derive an internal subclass of RegionIterator as well, although this is not necessary for file readers derived from StreamReader.

Figure 2 shows a region store implementation that reads regions from a simple text based format. The format contains a chromosome name, start and end locations and a unique name:

```
chr1 100 200 region1
chr1 300 450 region2
```

The reader is derived from StreamReader for demonstration, although CSVStreamReader could also be used due to additional convenience. The constructor sets up the annotation registry, which in this case contains one string-typed annotation. It is registered as a special kind of annotation, ANNOTATION_ID, to inform the framework that the values are unique identifiers. The method read_next reads one entry at a time, modifies an output object and returns true if a region was read. The file format uses 1-based coordinates: as Region uses 0-based coordinates, they must be converted in the reader. Region names are used to generate uniqueness tags for regions using a hash function. The static method create sets up a weak shared pointer for the newly created object.

```
#include <fstream>
#include <iostream>
#include <string>
#include "Region.hpp"
#include "RegionIterator.hpp"
#include "io/StreamReader.hpp"

class MyReader: public StreamReader {
public:
    static RegionStoreP create(const char *filename) {
        RegionStoreP store(new MyReader(filename));
        store->set_weak(store);
        return store;
    }
protected:
    MyReader(const char *filename) : input(filename) {
        this->name_id = add_annotation("Name",
            ANN_STRING, -1, ANNOTATION_ID);
    }
    bool read_next(Region& output, bool first) {
        std::string chr, name;
        int start, end;

        this->input >> chr >> start >> end >> name;
        if (this->input.eof()) return false;

        output.set_seqid(get_seqset().add(chr));
        output.set_left(start - 1);
        output.set_right(end - 1);
        output.set_annotationS(this->name_id, name);
        output.set_tag(name);
        return true;
    }
private:
    std::ifstream input;
    int name_id;
};

int main() {
    RegionStoreP reader(MyReader::create("custom-format.txt"));
    RegionIterator *iter = reader->iter();
    const Region *region;
    while ((region = iter->next()) != NULL) {
        std::cout << *region;
    }
    iter->close();
    return 0;
}
```

Figure 2: Region store that reads a simple text based format.