Salmon Documentation

Release 0.5.0

Rob Patro, Carl Kingsford and Steve Mount

June 03, 2016

Contents

Contents:

CHAPTER 1

Requirements

1.1 Binary Releases

Pre-compiled binaries of the latest release of Salmon for a number different platforms are available available under the Releases tab of Salmon's GitHub repository. You should be able to get started quickly byfinding a binary from the list that is compatible with your platform. If you're running an old version of Linux and get errors related to *GLIBC*, try the pre-compiled "Debian Squeeze" binary.

1.2 Requirements for Building from Source

- A C++11 conformant compiler (currently tested with GCC>=4.7 and Clang>=3.4)
- CMake. Salmon uses the CMake build system to check, fetch and install dependencies, and to compile and install Salmon. CMake is available for all major platforms (though Salmon is currently unsupported on Windows.)

CHAPTER 2

Installation

After downloading the Salmon source distribution and unpacking it, change into the top-level directory:

```
> cd salmon
```

Then, create and out-of-source build directory and change into it:

```
> mkdir build
> cd build
```

Salmon makes extensive use of Boost. We recommend installing the most recent version (1.55) systemwide if possible. If Boost is not installed on your system, the build process will fetch, compile and install it locally. However, if you already have a recent version of Boost available on your system, it make sense to tell the build system to use that.

If you have Boost installed you can tell CMake where to look for it. Likewise, if you already have Intel's Threading Building Blocks library installed, you can tell CMake where it is as well. The flags for CMake are as follows:

- -DFETCH_BOOST=TRUE – If you don't have Boost installed (or have an older version of it), you can provide the FETCH_BOOST flag instead of the BOOST_ROOT variable, which will cause CMake to fetch and build Boost locally.
- -DBOOST_ROOT=<boostdir> – Tells CMake where an existing installtion of Boost resides, and looks for the appropritate version in <boostdir>. This is the top-level directory where Boost is installed (e.g. /opt/local).
- -DTBB_INSTALL_DIR=<tbbroot> – Tells CMake where an existing installation of Intel's TBB is installed (<tbbroot>), and looks for the apropriate headers and libraries there. This is the top-level directory where TBB is installed (e.g. /opt/local).
- -DCMAKE_INSTALL_PREFIX=<install_dir> – <install_dir> is the directory to which you wish Salmon to be installed. If you don't specify this option, it will be installed locally in the top-level directory (i.e. the directory directly above "build").

There are a number of other libraries upon which Salmon depends, but CMake should fetch these for you automatically.

Setting the appropriate flags, you can then run the CMake configure step as follows:

```
> cmake [FLAGS] ..
```

The above command is the cmake configuration step, which *should* complain if anything goes wrong. Next, you have to run the build step. Depending on what libraries need to be fetched and installed, this could take a while (specifically if the installation needs to install Boost). To start the build, just run make.

```
> make
```

If the build is successful, the appropriate executables and libraries should be created. There are two points to note about the build process. First, if the build system is downloading and compiling boost, you may see a large number of warnings during compilation; these are normal. Second, note that CMake has colored output by default, and the steps which create or link libraries are printed in red. This is the color chosen by CMake for linking messages, and does not denote an error in the build process.

Finally, after everything is built, the libraries and executable can be installed with:

```
> make install
```

You can test the installation by running

```
> make test
```

This should run a simple test and tell you if it succeeded or not.

CHAPTER 3

Salmon

Salmon is a tool for **wicked-fast** transcript quantification from RNA-seq data. It requires a set of target transcripts (either from a reference or *de-novo* assembly) to quantify. All you need to run Salmon is a FASTA file containing your reference transcripts and a (set of) FASTA/FASTQ file(s) containing your reads. Optionally, Salmon can make use of pre-computed alignments (in the form of a SAM/BAM file) to the transcripts rather than the raw reads.

The **quasi-mapping**-based mode of Salmon runs in two phases; indexing and quantification. The indexing step is independent of the reads, and only need to be run one for a particular set of reference transcripts. The quantification step, obviously, is specific to the set of RNA-seq reads and is thus run more frequently. For a more complete description of all available options in Salmon, see below.

The **alignment**-based mode of Salmon does not require indexing. Rather, you can simply provide Salmon with a FASTA file of the transcripts and a SAM/BAM file containing the alignments you wish to use for quantification.

3.1 Using Salmon

As mentioned above, there are two "modes" of operation for Salmon. The first, requires you to build an index for the transcriptome, but then subsequently processes reads directly. The second mode simply requires you to provide a FASTA file of the transcriptome and a `.sam` or `.bam` file containing a set of alignments.

Note: Read / alignment order

Salmon, like eXpress, uses a streaming inference method to perform transcript-level quantification. One of the fundamental assumptions of such inference methods is that observations (i.e. reads or alignments) are made "at random". This means, for example, that alignments should **not** be sorted by target or position. If your reads or alignments do not appear in a random order with respect to the target transcripts, please randomize / shuffle them before performing quantification with Salmon.

Note: Number of Threads

The number of threads that Salmon can effectively make use of depends upon the mode in which it is being run. In alignment-based mode, the main bottleneck is in parsing and decompressing the input BAM file. We make use of the Staden IO library for SAM/BAM/CRAM I/O (CRAM is, in theory, supported, but has not been thorougly tested). This means that multiple threads can be effectively used to aid in BAM decompression. However, we find that throwing more than a few threads at file decompression does not result in increased processing speed. Thus, alignment-based Salmon will only ever allocate up to 4 threads to file decompression, with the rest being allocated to quantification. If these threads are starved, they will sleep (the quantification threads do not busy wait), but there is a point beyond which allocating more threads will not speed up alignment-based quantification. We find that allocating 8 — 12 threads results in the maximum speed, threads allocated above this limit will likely spend most of their time idle / sleeping.

For quasi-mapping-based Salmon, the story is somewhat different. Generally, performance continues to improve as more threads are made available. This is because the determiniation of the potential mapping locations of each read is, generally, the slowest step in quasi-mapping-based quantification. Since this process is trivially parallelizable (and well-parallelized within Salmon), more threads generally equates to faster quantification. However, there may still be a limit to the return on invested threads. Specifically, writing to the mapping cache (see *Misc* below) is done via a single thread. With a huge number of quantification threads or in environments with a very slow disk, this may become the limiting step. If you're certain that you have more than the required number of observations, or if you have reason to suspect that your disk is particularly slow on writes, then you can disable the mapping cache (`--disableMappingCache`), and potentially increase the parallelizability of quasi-mapping-based Salmon.

3.2 Quasi-mapping-based mode (including lightweight alignment)

One of the novel and innovative features of Salmon is its ability to accurately quantify transcripts using *quasi-mappings*. Quasi-mappings are mappings of reads to transcript positions that are computed without performing a base-to-base alignment of the read to the transcript. Quasi-mapping is typically **much** faster to compute than traditional (or full) alignments, and can sometimes provide superior accuracy by being more robust to errors in the read or genomic variation from the reference sequence.

Salmon currently supports two different methods for mapping reads to transcriptomes; (SMEM-based) lightweight-alignment and quasi-mapping. SMEM-based mapping is the original lightweight-alignment method used by Salmon, and quasi-mapping is a newer and considerably faster alternative. Both methods are currently exposed via the same `quant` command, but the methods require different indices so that SMEM-based mapping cannot be used with a quasi-mapping index and vice-versa.

If you want to use Salmon in quasi-mapping-based mode, then you first have to build an Salmon index for your transcriptome. Assume that `transcripts.fa` contains the set of transcripts you wish to quantify. First, you run the Salmon indexer:

```
> ./bin/salmon index -t transcripts.fa -i transcripts_index --type quasi -k 31
```

This will build the quasi-mapping-based index, using an auxiliary k-mer hash over k-mers of length 31. While quasi-mapping will make used of arbitrarily long matches between the query and reference, the *k* size selected here will act as the *minimum* acceptable length for a valid match. Thus, a smaller value of *k* may slightly improve sensitivty. We find that a *k* of 31 seems to work well for reads of 75bp or longer, but you might consider a smaller *k* if you plan to deal with shorter reads. Note that there is also a *k* parameter that can be passed to the `quant` command. However, this has no effect if one is using a quasi-mapping index, as the *k* value provided during the index building phase overrides any *k* provided during quantification in this case. Since quasi-mapping is the default index type in Salmon, you can actually leave off the `--type quasi` parameter when building the index. To build a lightweight-alignment (FMD-based) index instead, one would use the following command:

```
> ./bin/salmon index -t transcripts.fa -i transcripts_index --type fmd
```

Note that no value of *k* is given here. However, the SMEM-based mapping index makes use of a parameter *k* that is passed in during the `quant` phase (the default value is *19*).

Then, you can quantify any set of reads (say, paired-end reads in files *reads1.fq* and *reads2.fq*) directly against this index using the Salmon `quant` command as follows:

```
> ./bin/salmon quant -i transcripts_index -l <LIBTYPE> -1 reads1.fq -2 reads2.fq -o transcripts_quant
```

If you are using single-end reads, then you pass them to Salmon with the `-r` flag like:

```
> ./bin/salmon quant -i transcripts_index -l <LIBTYPE> -r reads.fq -o transcripts_quant
```

This same `quant` command will work with either index (quasi-mapping or SMEM-based), and Salmon will automatically determine the type of index being read and perform the appropriate lightweight mapping accordingly.

Note: Order of command-line parameters

The library type `-l` should be specified on the command line **before** the read files (i.e. the parameters to `-1` and `-2`, or `-r`). This is because the contents of the library type flag is used to determine how the reads should be interpreted.

You can, of course, pass a number of options to control things such as the number of threads used or the different cutoffs used for counting reads. Just as with the alignment-based mode, after Salmon has finished running, there will be a directory called `salmon_quant`, that contains a file called `quant.sf` containing the quantification results.

3.3 Alignment-based mode

Say that you've prepared your alignments using your favorite aligner and the results are in the file `aln.bam`, and assume that the sequence of the transcriptome you want to quantify is in the file `transcripts.fa`. You would run Salmon as follows:

```
> ./bin/salmon quant -t transcripts.fa -l <LIBTYPE> -a aln.bam -o salmon_quant
```

The `<LIBTYPE>` parameter is described below and is shared between both modes of Salmon. After Salmon has finished running, there will be a directory called `salmon_quant`, that contains a file called `quant.sf`. This contains the quantification results for the run, and the columns it contains are similar to those of Sailfish (and self-explanatory where they differ).

For the full set of options that can be passed to Salmon in its alignment-based mode, and a description of each, run `salmon quant --help-alignment`.

Note: Genomic vs. Transcriptomic alignments

Salmon expects that the alignment files provided are with respect to the transcripts given in the corresponding fasta file. That is, Salmon expects that the reads have been aligned directly to the transcriptome (like RSEM, eXpress, etc.) rather than to the genome (as does, e.g. Cufflinks). If you have reads that have already been aligned to the genome, there are currently 3 options for converting them for use with Salmon. First, you could convert the SAM/BAM file to a FAST{A/Q} file and then use the lightweight-alignment-based mode of Salmon described below. Second, given the converted FASTA{A/Q} file, you could re-align these converted reads directly to the transcripts with your favorite aligner and run Salmon in alignment-based mode as described above. Third, you could use a tool like sam-xlate to try and convert the genome-coordinate BAM files directly into transcript coordinates. This avoids the necessity of having to re-map the reads. However, we have very limited experience with this tool so far.

Multiple alignment files

If your alignments for the sample you want to quantify appear in multiple .bam/.sam files, then you can simply provide the Salmon `-a` parameter with a (space-separated) list of these files. Salmon will automatically read through these one after the other quantifying transcripts using the alignments contained therein. However, it is currently the case that these separate files must (1) all be of the same library type and (2) all be aligned with respect to the same reference (i.e. the @SQ records in the header sections must be identical).

3.4 Description of important options

Salmon exposes a number of useful optional command-line parameters to the user. The particularly important ones are explained here, but you can always run `salmon quant -h` to see them all.

3.4.1 `-p` / `--numThreads`

The number of threads that will be used for quasi-mapping, quantification, and bootstrapping / posterior sampling (if enabled). Salmon is designed to work well with many threads, so, if you have a sufficient number of processors, larger values here can speed up the run substantially.

3.4.2 `--useVBOpt`

Use the variational Bayesian EM algorithm rather than the "standard" EM algorithm to optimize abundance estimates. The details of the VBEM algorithm can be found in [2]_, and the details of the variant over fragment equivalence classes that we use can be found in [3]_. While both the standard EM and the VBEM produce accurate abundance estimates, those produced by the VBEM seem, generally, to be a bit more accurate. Further, the VBEM tends to converge after fewer iterations, so it may result in a shorter runtime; especially if you are computing many bootstrap samples.

3.4.3 `--numBootstraps`

Salmon has the ability to optionally compute bootstrapped abundance estimates. This is done by resampling (with replacement) from the counts assigned to the fragment equivalence classes, and then re-running the optimization procedure, either the EM or VBEM, for each such sample. The values of these different bootstraps allows us to assess technical variance in the main abundance estimates we produce. Such estimates can be useful for downstream (e.g. differential expression) tools that can make use of such uncertainty estimates. This option takes a positive integer that dictates the number of bootstrap samples to compute. The more samples computed, the better the estimates of varaiance, but the more computation (and time) required.

3.4.4 `--numGibbsSamples`

Just as with the bootstrap procedure above, this option produces samples that allow us to estimate the variance in abundance estimates. However, in this case the samples are generated using posterior Gibbs sampling over the fragment equivalence classes rather than bootstrapping. We are currently analyzing these different approaches to assess the potential trade-offs in time / accuracy. The `--numBootstraps` and `--numGibbsSamples` options are mutually exclusive (i.e. in a given run, you must set at most one of these options to a positive integer.)

3.4.5 `--biasCorrect`

Passing the `--biasCorrect` flag to Salmon will enable it to learn and correct for sequence-specific biases in the input data. Specifically, this model will attempt to correct for random hexamer priming bias, which results in the preferential sequencing of fragments starting with certain nucleotide motifs. By default, Salmon learns the sequence-specific bias parameters using 1,000,000 reads from the beginning of the input. If you wish to change the number of samples from which the model is learned, you can use the `--numBiasSamples` parameter. *Note*: This sequence-specific bias model is substantially different from the bias-correction methodology that was used in Salmon versions prior to 0.6.0 (and Sailfish versions prior to 0.9.0). This model specifically accounts for sequence-specific bias, and should not be prone to the over-fitting problem that was sometimes observed using the previous bias-correction methodology.

3.4.6 `--useFSPD`

Passing the `--useFSPD` flag to Salmon will enable modeling of a position-specific fragment start distribution. This is meant to model non-uniform coverage biases that are sometimes present in RNA-seq data (e.g. 5' or 3' positional bias). Currently, a single global model is learned and applied to all transcripts, as there is typically not enough information to learn a separate model for each transcript. However, modeling the effect in this manner can still be helpful when there is a global bias in coverage. In the future, we will potentially be exploring more fine-grained positional bias models.

3.5 What's this `LIBTYPE`?

Salmon, like sailfish, has the user provide a description of the type of sequencing library from which the reads come, and this contains information about e.g. the relative orientation of paired end reads. However, we've replaced the somewhat esoteric description of the library type with a simple set of strings; each of which represents a different type of read library. This new method of specifying the type of read library is being back-ported into Sailfish and will be available in the next release.

The library type string consists of three parts: the relative orientation of the reads, the strandedness of the library, and the directionality of the reads.

The first part of the library string (relative orientation) is only provided if the library is paired-end. The possible options are:

```
I = inward
O = outward
M = matching
```

The second part of the read library string specifies whether the protocol is stranded or unstranded; the options are:

```
S = stranded
U = unstranded
```

If the protocol is unstranded, then we're done. The final part of the library string specifies the strand from which the read originates in a strand-specific protocol — it is only provided if the library is stranded (i.e. if the library format string is of the form S). The possible values are:

```
F = read 1 (or single-end read) comes from the forward strand
R = read 1 (or single-end read) comes from the reverse strand
```

An example of some library format strings and their interpretations are:

```
IU (an unstranded paired-end library where the reads face each other)
```

```
SF (a stranded single-end protocol where the reads come from the forward strand)
```

```
OSR (a stranded paired-end protocol where the reads face away from each other,
     read1 comes from reverse strand and read2 comes from the forward strand)
```

Note: Strand Matching

Above, when it is said that the read "comes from" a strand, we mean that the read should align with / map to that strand. For example, for libraries having the `OSR` protocol as described above, we expect that read1 maps to the reverse strand, and read2 maps to the forward strand.

For more details on the library type, see *Fragment Library Types*.

3.6 Output

For details of Salmon's different output files and their formats see :ref: *FileFormats*.

3.7 Misc

Salmon deals with reading from compressed read files in the same way as sailfish — by using process substitution. Say in the lightweigh-alignment-based salmon example above, the reads were actually in the files `reads1.fa.gz` and `reads2.fa.gz`, then you'd run the following command to decompress the reads "on-the-fly":

```
> ./bin/salmon quant -i transcripts_index -l <LIBTYPE> -1 <(gzcat reads1.fa.gz) -2 <(gzcat reads2.fa
```

and the gzipped files will be decompressed via separate processes and the raw reads will be fed into salmon.

Finally, the purpose of making this software available is for people to use it and provide feedback. The pre-print describing this method is on bioRxiv. If you have something useful to report or just some interesting ideas or suggestions, please contact us (*rob.patro@cs.stonybrook.edu* and/or *carlk@cs.cmu.edu*). If you encounter any bugs, please file a *detailed* bug report at the Salmon GitHub repository.

Salmon Output File Formats

4.1 Quantification File

Salmon's main output is its quantification file. This file is a plain-text, tab-separated file with a single header line (which names all of the columns). This file is named `quant.sf` and appears at the top-level of Salmon's output directory. The columns appear in the following order:

Name	Length	EffectiveLength	TPM	NumReads

Each subsequent row describes a single quantification record. The columns have the following interpretation.

- **Name** — This is the name of the target transcript provided in the input transcript database (FASTA file).
- **Length** — This is the length of the target transcript in nucleotides.
- **EffectiveLength** — This is the computed *effective* length of the target transcript. It takes into account all factors being modeled that will effect the probability of sampling fragments from this transcript, including the fragment length distribution and sequence-specific and gc-fragment bias (if they are being modeled).
- **TPM** — This is salmon's estimate of the relative abundance of this transcript in units of Transcripts Per Million (TPM). TPM is the recommended relative abundance measure to use for downstream analysis.
- **NumReads** — This is salmon's estimate of the number of reads mapping to each transcript that was quantified. It is an "estimate" insofar as it is the expected number of reads that have originated from each transcript given the structure of the uniquely mapping and multi-mapping reads and the relative abundance estimates for each transcript.

4.2 Command Information File

In the top-level quantification directory, there will be a file called `cmd_info.json`. This is a JSON format file that records the main command line parameters with which Salmon was invoked for the run that produced the output in this directory.

4.3 Auxiliary File

The top-level quantification directory will contain an auxiliary directory called `aux` (unless the auxiliary directory name was overridden via the command line). This directory will have a number of files (and subfolders) depending on how salmon was invoked.

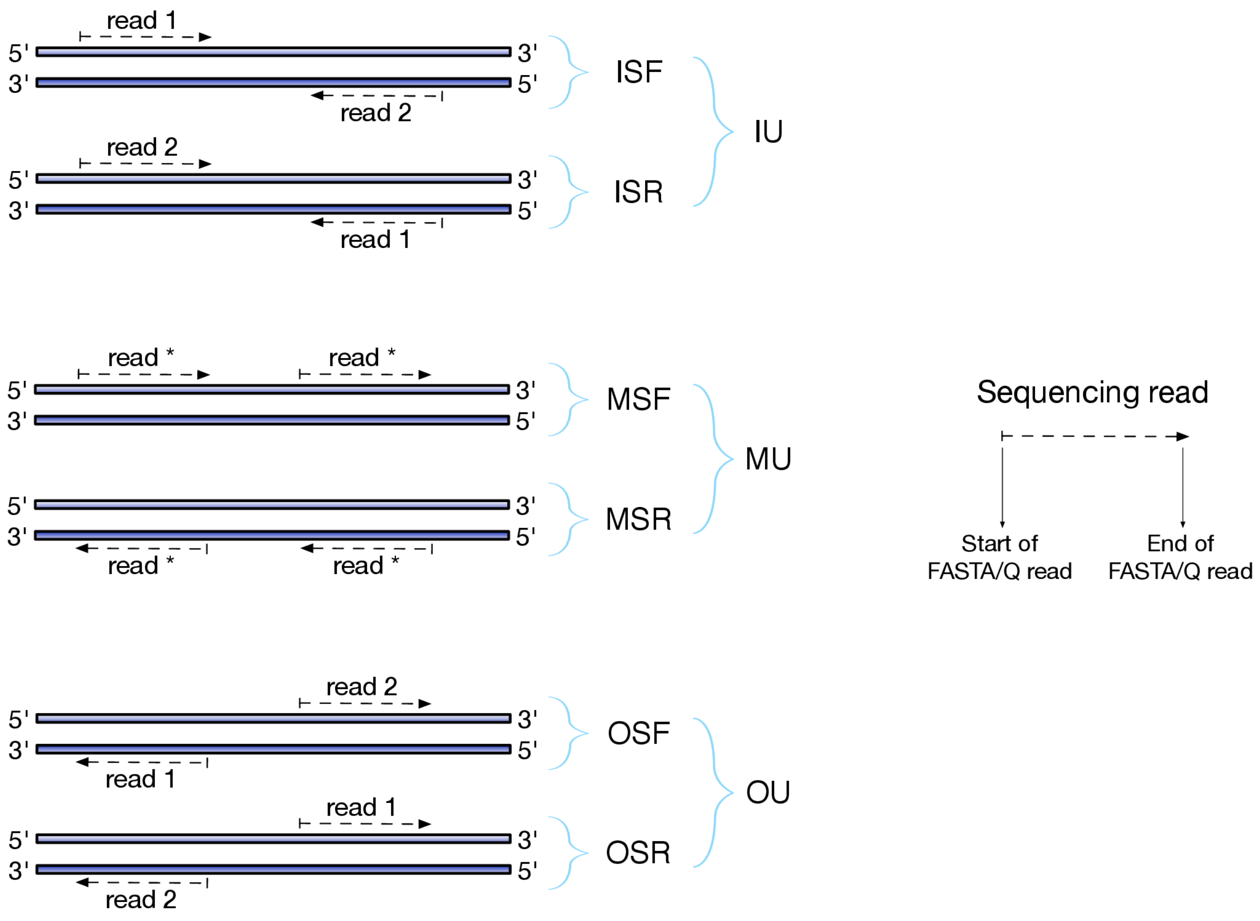
4.3.1 `fld.gz`

This file contains an approximation of the observed fragment length distribution. It is a gzipped, binary file containing integer counts. The number of (signed, 32-bit) integers (with machine-native endianness) is equal to the number of bins in the fragment length distribution (1,001 by default — for fragments ranging in length from 0 to 1,000 nucleotides).

CHAPTER 5

Fragment Library Types

There are numerous library preparation protocols for RNA-seq that result in sequencing reads with different characteristics. For example, reads can be single end (only one side of a fragment is recorded as a read) or paired-end (reads are generated from both ends of a fragment). Further, the sequencing reads themselves may be unstranded or strand-specific. Finally, paired-end protocols will have a specified relative orientation. To characterize the various different typs of sequencing libraries, we've created a miniature "language" that allows for the succinct description of the many different types of possible fragment libraries. For paired-end reads, the possible orientations, along with a graphical description of what they mean, are illustrated below:



The library type string consists of three parts: the relative orientation of the reads, the strandedness of the library, and the directionality of the reads.

The first part of the library string (relative orientation) is only provided if the library is paired-end. The possible options are:

```
I = inward
O = outward
M = matching
```

The second part of the read library string specifies whether the protocol is stranded or unstranded; the options are:

```
S = stranded
U = unstranded
```

If the protocol is unstranded, then we're done. The final part of the library string specifies the strand from which the read originates in a strand-specific protocol — it is only provided if the library is stranded (i.e. if the library format string is of the form S). The possible values are:

```
F = read 1 (or single-end read) comes from the forward strand
R = read 1 (or single-end read) comes from the reverse strand
```

So, for example, if you wanted to specify a fragment library of strand-specific paired-end reads, oriented toward each other, where read 1 comes from the forward strand and read 2 comes from the reverse strand, you would specify `-l ISF` on the command line. This designates that the library being processed has the type "ISF" meaning, **I**nward (the relative orientation), **S**tranted (the protocol is strand-specific), **F**orward (read 1 comes from the forward strand).

The single end library strings are a bit simpler than their pair-end counter parts, since there is no relative orientation of which to speak. Thus, the only possible library format types for single-end reads are `U` (for unstranded), `SF` (for strand-specific reads coming from the forward strand) and `SR` (for strand-specific reads coming from the reverse strand).

A few more examples of some library format strings and their interpretations are:

```
IU (an unstranded paired-end library where the reads face each other)
```

```
SF (a stranded single-end protocol where the reads come from the forward strand)
```

```
OSR (a stranded paired-end protocol where the reads face away from each other,
     read1 comes from reverse strand and read2 comes from the forward strand)
```

Note: Correspondence to TopHat library types

The popular TopHat RNA-seq read aligner has a different convention for specifying the format of the library. Below is a table that provides the corresponding sailfish/salmon library format string for each of the potential TopHat library types:

TopHat	Salmon (and Sailfish)	
	Paired-end	Single-end
`-fr-unstranded`	`-l IU`	`-l U`
`-fr-firststrand`	`-l ISR`	`-l SR`
`-fr-secondstrand`	`-l ISF`	`-l SF`

The remaining salmon library format strings are not directly expressible in terms of the TopHat library types, and so there is no direct mapping for them.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search