# Ensembl BlastView: Search API

William Spooner (whs@sanger.ac.uk)

4 September 2003

## Contents



## 1 Introduction

This document provides details of a BioPerl-based API used for running, parsing and storing sequence similarity searches. This API was developed in support of the Ensembl BlastView web interface (http://www.ensembl.org/Multi/blastview). However, due to the componentised nature of the code, its constituent parts can easily be incorperated into other software projects.

There are several main components of the Search API;

- Search report parsing,

- Search report storage,
- Running searches to generate reports.

The architecture of the Search API is based closely on BioPerl's IO factory system, and extensive use is made of the SearchIO system in particular. In addition to increasing the functional scope of BioPerl's API, Ensembl-specific extensions to existing BioPerl classes have been developed. These integrate Ensembl API classes into the object representations of search reports, and provide potential for annotation beyond the limitations of sequence databases in isolation.

This document describes the technical details of the Search API. Code examples, writen in Perl, are included.

## 2 Software availability and Layout

The Ensembl Web code is available under an open source license from:

```
http://www.ensembl.org
```

The BioPerl library is integral to the BlastView system. It is also open source, available from:

```
http://www.bioperl.org
```

Full details on installing the Ensembl web site are available for download from:

```
http://www.ensembl.org/Docs/wiki/html/EnsemblDocs/InstallEnsemblWebsite.html
```

The Ensembl Web distribution is usually stored within a filesystem directory (hereafter refered to as `$ENSEMBL_ROOT`), e.g.:

```
$ setenv ENSEMBL_ROOT /usr/local/ensembl
```

The Search API is organised across several library directories within the Ensembl web distribution, including:

| | |
|---|---|
| `$ENSEMBL_ROOT/modules` | The Ensembl Web API library |
| `$ENSEMBL_ROOT/ensembl/modules` | The core Ensembl API library |
| `$ENSEMBL_ROOT/bioperl-live` | The BioPerl library |

# 3 Parsing search reports

## 3.1 Bio::SearchIO

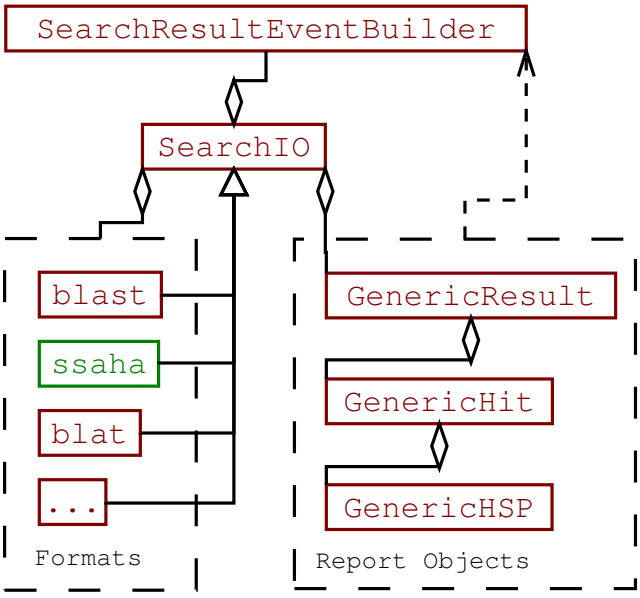
Bio::SearchIO is an integral part of the BioPerl library, used for parsing search reports. It's action is to convert a given ASCII search report into a heirarchy of Perl objects; Results (one per query sequence), Hits (database sequences that align with the query sequence), and HSPs (High Scoring Pairs; the alignment of the query and hit sequence). Result, Hit and HSP objects have a common interface regardless of the format of the original search report.

Having generated a search report using, for example, wu-blastn, SearchIO is typically run as:

```
use Bio::SearchIO;
my $sio = Bio::SearchIO->new( -format=>'blast',
                              -file=>'report.bls' );
while( my $res = $sio->next_result ){
  while( my $hit = $res->next_hit ){
    while( my $hsp = $hit->next_hsp ){
      # Display alignment
    }
  }
}
```

A UML representation of SearchIO is included as figure 1.

Figure 1: *A UML representation of the SearchIO framework. Classes in red are located in the `£ENSEMBL_ROOT/bioperl-live` library, those in green are located in `£ENSEMBL_ROOT/modules`.*




Further documentation on Bio::SearchIO can be found at:

```
http://bioperl.org/HOWTOs/pdf/SearchIO.pdf
http://docs.bioperl.org/releases/bioperl-1.2.2/
```

## 3.2 Adding formats to Bio::SearchIO

SearchIO parses search reports using an event-based system to seperate the data into Result, Hit and HSP components (the SearchResultEventBuilder class in figure 1). Events are paired start/end calls, and include, for instance; start_result, end_result, start_hit etc. Differences in report formats are handled by format-specific Perl modules located in:

```
$ENSEMBL_ROOT/bioperl-live/Bio/SearchIO/
```

For formats where there are no standard parsers (e.g. SSAHA), a new modules have been developed, and are stored within the $ENSEMBL_ROOT/modules Perl library (see the 'ssaha' format class in figure 1). For example:

```
$ENSEMBL_ROOT/modules/Bio/SearchIO/ssaha.pm
```

## 3.3 Extending Bio::SearchIO with the Ensembl API

By default, SearchIO generates objects blessed into the GenericResult class. SearchIO can, however, be configured to generate objects of any class, so long as they inheret from, and implement, a standard interface (Bio::Search::Result::ResultI). Similar logic and interfaces exist for both Hit and HSP objects. For BlastView, three new classes have been developed that inheret from the base interfaces; EnsemblResult, EnsemblHit and EnsembHSP. A UML representation of these new objects is included as figure 2.

SearchIO is configured to generate results of a different class as follows:

```
$class = 'Bio::Search::Result::EnsemblResult';
$sio = Bio::SearchIO->new(-file=>$file,
                          -format=>$format);
$fact = Bio::Search::Result::ResultFactory->new(-type=>$class);
$sio->_eventHandler->register_factory($fact);
```

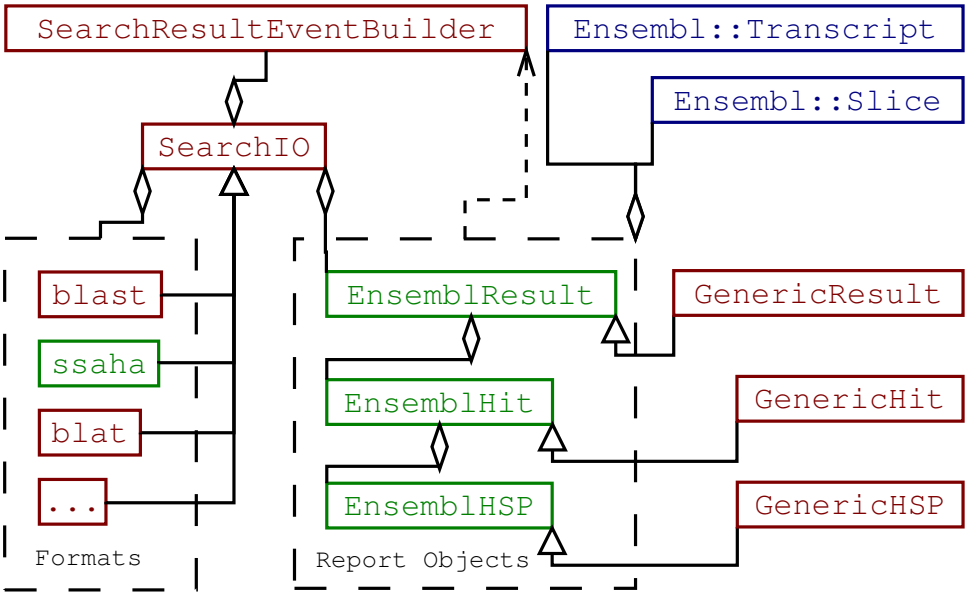
EnsemblResult, EnsemblHit and EnsembHSP differ from the generic objects in that they provide access to Ensembl API objects (Slice, Transcript etc) relevant to the database being searched against. At present, the Ensembl API is used at the parsing stage to map alignment locations from database coordinates into genomic coordinates. For searches against contig databases, 'non-golden' alignments are removed at this stage.

Future developments will make further use of the Ensembl API to provide value-add information to the search results, e.g. upstream/downstream sequences, genomic alignments etc..

The extended SearchIO modules developed for BlastView are:

```
$ENSEMBL_ROOT/modules/Bio/Search/Result/EnsemblResult.pm
$ENSEMBL_ROOT/modules/Bio/Search/Hit/EnsemblHit.pm
$ENSEMBL_ROOT/modules/Bio/Search/HSP/EnsemblHSP.pm
```

Figure 2: *UML of the BlastView extensions to the SearchIO framework. Classes in red are located in the* `£ENSEMBL_ROOT/bioperl-live` *library, those in green are located in* `£ENSEMBL_ROOT/modules`*, and those in blue are loacted in* `£ENSEMBL_ROOT/ensembl/modules`*.*




# 4 Storing Search Reports

## 4.1 Bio::Root::Storable

Parsing search reports with SearchIO is, in the context of web applications, a memory intensive and time consuming process. As BlastView is a multi-page application it is convenient to parse the report once, store its parsed representation to disk, and retrieve this for each page request. A string representation of a Perl object can be created/recovered using the standard Storable module. This forms the basis of a new BioPerl root module, Bio::Root::Storable. This is a generic module, which can be inhereted by any BioPerl object through inclusion in the @ISA array. By default, Bio::Root::Storable serialises objects to the system temporary directory (/tmp on UNIX/Linux):

```
$obj = Bio::MyObj->new(); # Bio::MyObj inherets from Bio::Root::Storable
$token = $obj->store(); # Returns the name of the temporary file
$obj2 = Bio::MyObj->retrieve( $token ); # Returns a clone of $obj
```

The default behaviour of Bio::Root::Storable can be altered by specifying; the directory to store state files, a filename template, and/or a filename suffix. For example:

```
$obj = Bio::MyObj->new(-workdir=>'/tmp/foo',
                       -template=>'BLA_XXXXXXXX',
                       -suffix=>'.obj');
print $obj->store(); # Should print e.g.; /tmp/foo/BLA_jsS20Iok.obj
```

In addition to store/retrieve, objects inhereting from Bio::Root::Storable can be collapsed and expanded. The collapsed object (created with the 'new_retrievable' call; should this be renamed 'collapse'?) contains nothing except the token indicating where the state is stored. This object can be re-populated later using the 'retrieve' call (should this be renamed 'expand'):

```
$obj2 = $obj->new_retrievable; # $obj2 is a skeleton object
if( $stub->retrievable ){ $obj2->retrieve } # $obj2 is now a clone of $obj
```

Storable objects can also be removed explicitly...(more on clean-up including auto-clean):

```
$obj->remove;
```

The main use of new_retrievable/retrieve is for recursive storage of object heirarchies (e.g. parsed search reports), and subsequent lazy-retrieve. When the 'store' method is called on a parent object, the data structure is examined. If child objects are found, they are stored using the 'new_retrievable' method. I.e. the parent stores the skeleton representation of the child, rather than the child itself. When the parent is later retrieved, its children remain in the skeleton state until explicitly retrieved by the parent. This design has obvious efficiency gains w.r.t. memory usage; important for large search reports.

The Bio::Root::Storable module is located at:

```
$ENSEMBL_ROOT/modules/Bio/Root/Storable.pm
```

## 4.2 Extending Bio::Root::Storable for Database Access

Whilst Bio::Root::Storable provides adequate tools for saving objects to the file system, it is often convenient to save data using alternatives, such as in a relational database. This is easily achieved on a per-module basis by overriding the Bio::Root::Storable->store/retrieve calls, as implemented by the EnsemblResult, EnsemblHit and EnsemblHSP modules. Following the Ensembl API model, all database read/write is mediated through an adaptor module, Bio::EnsEMBL::External::BlastAdaptor:

```
$ENSEMBL_ROOT/ensembl/modules/Bio/EnsEMBL/External/BlastAdaptor.pm
```

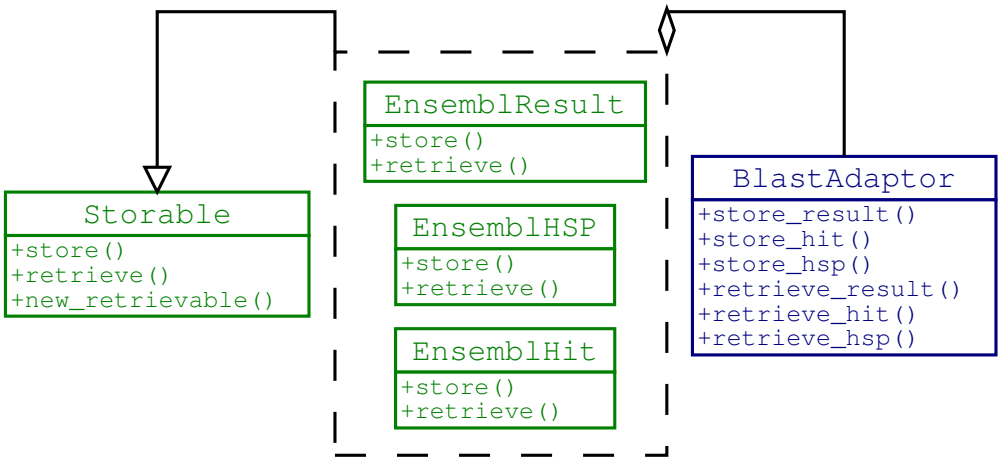
For instance, the EnsemblResult 'store' method uses the BlastAdaptor as follows:

```
sub store{
  my $self = shift;
  return $self->blast_adaptor->store_result( $self );
}
```

Subsequently, the BlastAdaptor 'store_result' generated the serialised object, and runs the SQL used to insert/update the object in the database. The return value from EnsemblResult 'store' is the database result_id rather than a filename. This is also the token required by the EnsemblResult 'retrieve' method. Note also that the BlastAdaptor must be passed to the EnsemblResult object explicitly during object instantiation, or using the blast_adaptor() method before either 'store' or 'retrieve' methods are called.

The procedure for store/retrieve of EnsemblHit and EnsemblHSP objects is similar to that outlined above. It is permitted, however, for certain objects in a heirarchy to be saved to the filesystem, whilst others are saved to a database. A UML representation of BlastView's Storable implementation of included as figure 3.

Figure 3: *UML of BlastView's Storable implementation. Classes in green are located in the £ENSEMBL_ROOT/modules library and those in blue are loacted in £ENSEMBL_ROOT/ensembl/modules.*




## 4.3 The Ensembl BlastBiew database

To allow objects to be written to a database, a database schema is required. The database schema itself can be written directly from the BlastAdaptor module. Firstly, an empty database should be created (although the schema can be written to an existing database if required. The following code is then required:

```
my $ab_adpt = Bio::EnsEMBL::External::BlastAdaptor->new(
   -dbname => 'ensembl_blastview',
   -user   => 'admin_user',
   -pass   => 'secret',
   -host   => 'localhost',
   -port   => '3307' );
$blast_adaptor->create_tables();
$blast_adaptor->rotate_daily_tables();
```

There are several types of table in the Ensembl BlastView database. Firstly, there are the blast_result, blast_hit and blast_hsp object tables. These store representations of the Result, Hit and HSP objects as parsed from search reports. There is currently no explicit linking between the object tables; these links are maintained by the serialised objects themselves. The table names are appended with a date suffix. This means that old searches in the blast database can be removed by dropping tables, rather than by using row-level selective deletes. This approach is ideal for busy sites like Ensembl, where database maintenance schemes must avoid locking tables.

In addition to the object tables, there is a blast_table_log table which records the names of all object tables against their status ('CURRENT', 'FILLED', 'DELETED'). Whilst new searches are written to the 'CURRENT' object tables, those stored in 'FILLED' tables are still available for retrieval. The object tables are created and the blast_table_log updated using the ' rotate_daily_tables' BlastAdaptor method.

A final table, 'blast_ticket' is used for the grouping together of multiple searches against session tickets. The 'create_tables' method creates both the blast_table_log and blast_ticket tables if they do not already exist.

A Perl script is provided with the Ensembl web distribution that runs 'create_tables' and 'rotate_daily_tables' based on the BlastView database configured in the Ensembl web config (covered later):

```
$ENSEMBL_ROOT/utils/blast_database.pl
```

A further script is provided for rotating daily tables, and dropping daily tables older than a threshold number of days. This script is generally used from cron for automatic BlastView database maintenance:

```
$ENSEMBL_ROOT/utils/blast_cleaner.pl
```

# 5 Running Searches

## 5.1 Bio::Tools::Run::Search

The preceeding discussion has dealt with parsing and handling of existing search reports. To support BlastView, a new system has been developed that generates reports by running searches of query sequences against sequence databases. This system is based on the same model as SearchIO, and provides a consistent interface for a number of search methods (e.g. WU-BLAST, SSAHA etc).

The Search object must be initialised with a method string (see 'Adding Methods' section below), a method-specific database-location string (typically the filesystem path to a blast database, or the server:port of a SSAHA server), and a Bio::SeqI compliant object holding the sequence to be searched.

Searching a single database with a single query sequence is typically coded as:

```
$search = Bio::Tools::Run::Search(-method=>'ensembl_wublastn');
$search->database($database); # Database location string
$search->seq($seq); #Bio::SeqI compliant object
$search->run; # Launch the query
while( $search->status eq 'RUNNING' ){
  wait( 10 )
}
if( $search->status ne COMPLETED' ){
  die( 'Something wrong: ', $search->status )
}
$res = $search->result;  # Get the ResultI compliant object
while( $hit = $res->next_hit ){
  while( $hsp = $hit->next_hsp ){}
}
```

It should also be noted that the Search module implements a copy constructor. This means that existing Search objects can be cloned. This allows, for example, the method parameters to be changed without changing anything else. E.g.:

```
$new_search = $old_search->new();
$new_search->parameter(filter=>'dust');
```

As Search inherets from Storable, instances can safely be serialised as described above. At the moment the BlastView system uses the default Storable implemenation, and serialisation is therefore to the filesystem rather than to database. It is likely, however, that a subclass of Storable will be implemented for database storage (using BlastAdaptor) in a similar manner to the EnsemblResult class.

A UML diagram of the Search class and associated modules in shown in figure 4. The Bio::Tools::Run::Search module is located at:

```
$ENSEMBL_ROOT/modules/Bio/Tools/Run/Search.pm
```

## 5.2 Adding Methods to Bio::Tools::Run::Search

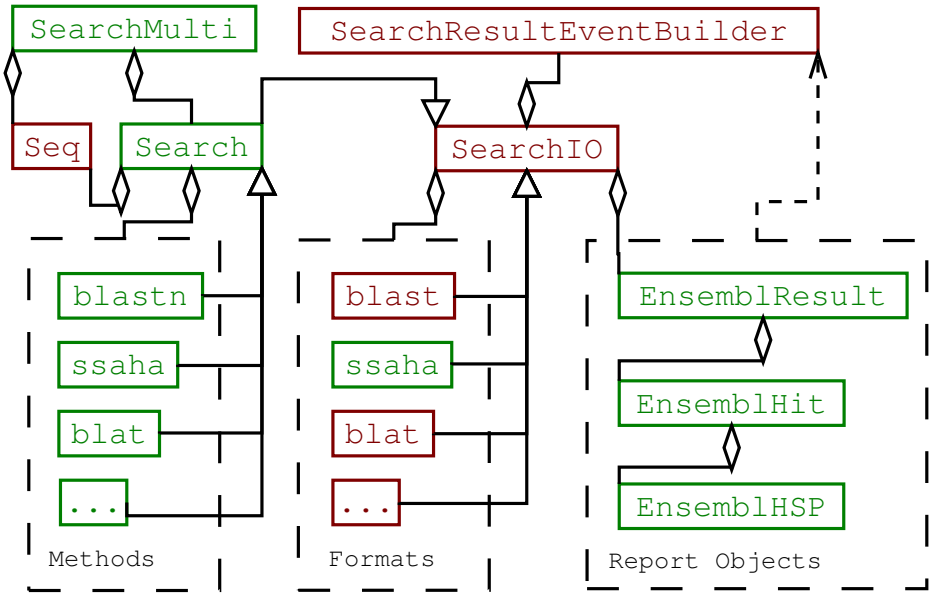
Methods available for use with Search must have a corresponding module in the Search sub-directory:

```
$ENSEMBL_ROOT/modules/Bio/Tools/Run/Search/
```

These modules control the way that method commands are generated, how the commands are dispatched, and how the results are retrieved. Creating a new Search object causes the module corresponding to the 'method' argument to be lazy-loaded. Developers can therefore create new modules to reflect their local systems/preferences without having to modify existing code. Modules that may be useful to the wider community (e.g. incorperation of different executables or dispatch systems), will gladly be accepted into the core code-base.

Existing methods include:

Figure 4: *UML diagram of the Search class and associated modules. It's realtionship to SearchIO is also shown. Classes in red are located in the* `£ENSEMBL_ROOT/bioperl-live` *library, those in green are located in* `£ENSEMBL_ROOT/modules`




```
ensembl_wublastn.pm ensembl_wublastx.pm  ensembl_wutblastx.pm
ensembl_wublastp.pm ensembl_wutblastn.pm ensembl_ssaha.pm
```

These modules all inheret from Bio::Tools::Run::Search, and implement the 'run' method. The differences between ensembl_wublast* methods are minor, and common code is abstrated into a seperate module (EnsemblBlast). The 'run' method is typically comprised of '_gen_command' and '_dispatch' methods.

Simple methods such as ensembl_ssaha have a '_gen_command' method that returns a string representing the shell command used to run the SSAHA client. Next, the '_dispatch' method is called which uses Perl's 'system' command to fork and run the command, waiting for the call to return. Finally, the run command parses the search report, updates the object's status attribute to 'COMPLETED', and returns. I.e. ensembl_ssaha's run method is an inline implementation, and the results are available for analysis in near real time.

The above scheme is not suitable, however, for methods like wu-blastn which may take an appreciable time to run. For the ensembl_wublast* methods, therefore, a different (offline) approach is taken. A basic example of the offline approach is to generate the blast command, dispatch the command using, for example, a job submission system such as BSUB, update the object status to 'RUNNING' and return. Report parsing must then wait until the report is available, whereupon it is parsed inline. For greatest efficiency, however, it is desirable to perform as much processing as possible during the offline phase. Such processing includes, for example, repeat-masking query sequences, and parsing the search report. To achieve this, the 'run' call dispatches a perl wrapper script rather than the blast command itself. This wrapper then retrieves the Search (which is a Storable object) and calls a seperate command

(e.g. 'run_offline' ), which prepares and dispatches the blast, and finally parses the report, in a manner similar to that for inline methods. At present, all BlastView WU-BLAST jobs are run using this offline warpper approach, with the perl script located at:

```
$ENSEMBL_ROOT/utils/run_blast.pl
```

Investigation into abstracting the method dispatch logic into a seperate object class is underway. Such classes would form a reusable component, and allow method modules to work with a variety of dispatchers. At the moment, a new method module is required for each dispatcher.

In addition to the 'run' method, each Search module requires a 'format' method; this must return the appropriate SearchIO format for parsing the search report; e.g. the ensembl_wublastn method has a format of 'blast', and ensembl_ssaha's format is 'ssaha'.

Note: the method module implementations for BlastView currently require direct access to the Ensembl web configuration data. Work is underway to remove such dependencies, with configuration via object methods providing an alternative (deployment independent) initialisation approach.

## 5.3 Multiple Searches with Bio::Tools::Run::SearchMulti

The Bio::Tools::Run::SearchMulti is a wrapper for generating and managing multiple Bio::Tools::Run::Search objects. Multiple methods, databases and sequences can be added to the container. It's 'run' method causes Search objects to be created from all methods vs. all databases vs. all sequences, and the 'run' method on seach to be called. An internal register of all methods, sequences and databases vs. Search objects is kept, allowing additional sequences etc to be added at any time. In addition it the 'add_...' methods, there exist corresponding 'delete_...' methods for clearing e.g. Search(es) for a given sequence from the register. Searches can be retrieved from the SearchMulti container in a number of ways, e.g.; all (via the 'runnables' method), all for a given Sequence (via 'runnables_like' with the '-seq' arg), or a a single named Search (via the ' runnables_like' with the '-ticket' arg). All of the Search fetching methods will retreive serialised Search objects on-demand, thereby minimising memory usage.

This module is typically used as follows;

```
$runmulti = Bio::Tools::Run::SearchMulti->new();
$runmulti->add_method($method);    # Search object (template)
$runmulti->add_database($database);# Database string
$runmulti->add_seq($seq);          # Bio::SeqI object
# Get a list of Bio::Tools::Run::Search objects
@runnables = grep{ $_->run } $runmulti->runnables;
```

The major limitation of SearchMulti is the all-against-all Search creation w.r.t. sequences, databases and methods. This behaviour is likely to be modified in the future to allow for e.g. databases being tied to specific methods etc.

## 5.4 Extending Bio::Tools::Run::SearchMulti

Bio::Tools::Run::SearchMulti uses the default Storable implementation whereby objects are serialised to the filesystem. A new class, EnsemblSearchMulti extends SearchMulti to allow objects to be serialised to the Ensembl BlastView database (using the BlastAdaptor described earlier). In addition, a multi-level directory heirarchy has been implemented for the storage of attached Search objects to avoid aproaching filesystem directory limitations.